AFIT/GCS/ENG/92D-03

SPATIAL PARTITIONING OF A BATTLEFIELD
PARALLEL DISCRETE-EVENT SIMULATION

THESIS

Kenneth C. Bergman
Captain, USAF

AFIT/GCS/ENG/92D-03

Approved for public release; distribution unlimited

AFIT/GCS/ENG/92D-03

SPATIAL PARTITIONING OF A BATTLEFIELD PARALLEL DISCRETE-EVENT SIMULATION

THESIS

Presented to the Faculty of the School of Engineering

of the Air Force Institute of Technology

Air University

In Partial Fulfillment of the

Requirements for the Degree of

Master of Science in Computer Engineering

Kenneth C. Bergman, B.S.C.S

Captain, USAF

December, 1992

Approved for public release; distribution unlimited

Acknowledgements

First of all, I want to thank my thesis committee — Major Eric Christensen, Dr. Thomas Hartrum, Dr. Gary Lamont, and Major Mike Garrambone — for all of their advice and helping me to perservere when I got bogged down by some aspect of my thesis effort. Major Christensen and Dr. Hartrum were especially invaluable in this regard; I really appreciated all their time and patience. Just think, Major Christensen, you can have 1Lt Guanu code his thesis effort in Ada to circumvent all the "shortcomings" that the 'C' language had in my research!

I also want to give a hearty thanks to Mr. Rick Norris. Whenever I ran up against those never-ending pointer errors, he was always willing to help me untangle them, as well as just be a friend. Fortunately my thesis quarter didn't have too many Hypercube failures due to thunderstorms; but whenever the system did fail, Rick came in and got it running again as soon as possible.

Last, but certainly not least, I want to thank my family. My wife Karen found it very hard to watch me struggle time and again with my research, knowing that all she could do was keep my spirits up and keep my son occupied so I could study. Even when I was physically home, she pointed out that I often had my head in the clouds thinking about school and my thesis; she coined the term "AFIT-brain" for my apparent malady. And my 2 $\frac{1}{2}$ year-old son Matt, he just couldn't understand why "Daddy" couldn't stay home at night and play with him all the time like we used to do. Whenever I headed for school, his chants of "Daddy Work" echoed behind me as he waved from the bedroom window. I know that both of them had to endure as much as I did, in their own way. Thank you for hanging in there with me. I love you both with all my heart.

Kenneth C. Bergman

Table of Contents

List of Figures

Figure		Page

List of Tables

AFIT/GCS/ENG/92D-03

Abstract

This thesis describes a method for spatially partitioning a battlefield into units known as sectors to achieve speedup two ways: through the reduction of each battlefield object's next event search space, and lowering the amount of message-passing required. Each sector is responsible for tracking and controlling access to all objects within its boundaries. A distributed proximity detection algorithm employing boundary-crossing events is used to control player movement between sectors.

Each object's state information is replicated in all sectors it has sensor capability for the minimum time required; this ensures that the object's next event is properly determined based upon interactions with objects in other sectors as well as its own.

Each scenario is initialized using three sources of information: a set of scenario input files which describe the battlefield and its objects, a mapping file which relates each sector to a particular logical process (LP), and command-line arguments which specify the files to use. During execution, each object travels along a set of prescripted route points — either attacking detected enemy objects with missiles or evading them — until it has either reached its destination route point or been destroyed.

Scenarios generate output in the form of screen messages, log files, and graphics display files which can be activated or deactivated at the user's discretion.

The issues involved in determining when and how to dynamically change the boundaries are discussed. A heuristic for changing sector boundaries based upon the number of players in each sector, as well as player attributes, is proposed.

SPATIAL PARTITIONING OF A
BATTLEFIELD PARALLEL DISCRETE-EVENT SIMULATION

I. Introduction

In March 1990, the United States Department of Defense (DoD) released a report identifying twenty technologies vital to ensuring the long-term qualitative superiority of our military forces — simulation and modeling technology (SMT) was one of them. SMT was categorized as one of the most valuable technologies available today, and was characterized as offering immediate advancement in DoD weapons systems capabilities (32:1). In the past, the United States Department of Defense (DoD) has relied heavily on the use of large, multi-service military exercises to maintain troop combat readiness. However, these exercises are very expensive, take too long to plan and execute, and can be perceived by foreign nations as being provocative. During the last twenty years, the United States has developed an extensive collection of computer simulations to take the place of exercises in the field. Computer simulators — usually manned by one person and executing sequentially — are excellent for training soldiers to do their jobs as individuals or as members of a small unit. However, the United States learned in Grenada, Panama, and Libya that the ability to perform a mission as an individual does not guarantee the ability to function as a member of a coordinated task force (20:1).

In February 1992, a draft paper was released describing an extension to the simulation network (SIMNET) developed by the Defense Advanced Research Projects Agency (DARPA)(21). This new network, known as Distributed Interactive Simulation (DIS), was designed primarily to support DoD training by allowing computer simulators spread out over a large geographical area to interact in a team environment. In order for DIS to succeed, operational guidelines and standards for inter-operability between the involved simulators have to be generated, potentially making it necessary to redesign some simulators to meet these new guidelines. As these simulators are redesigned and new ones are

developed in the future, it is vital that they maintain the ability to execute in a timely manner dependent upon the user's decision-making requirements.

Initially, DIS simulators are expected to come from existing simulators which usually have responsibility for only one object, with a "person in the loop" providing the intelligence which drives it. In the latter stages of the implementation of DIS, simulators are envisioned to become responsible for performing more tasks without personnel interaction. These tasks include the management of several similar objects in combat-level force structures, and the replacement of the "person in the loop" with machine intelligence to drive those objects. The objects controlled by a simulator with these qualities are referred to as computer-generated forces. In the future, the need for timely computer-generated forces can only be achieved through the use of parallel computer simulations.

Battlefield simulations are among the most irregular, computationally intensive, and complex simulations in existence, often taking days or weeks of computer time to complete just one scenario (40:14). Such long turnaround times severely impact the design cycle and often lead to sub-optimal designs (42:1). To meet the need of commanders and their staff, faster methods for conducting battlefield simulations are needed. Parallel computers offer the potential of an n-times speedup with an n-processor computer.

1.1 Background.

1.1.1 Role of Computer Simulation in DoD. Computer simulation is simply the use of a computer program to "mimic" the behavior of a proposal or existing system and thereby gain insight into the performance of that system under a variety of circumstances. Simulations are often used to determine how some aspect of a system should be designed, set up or operated (36:5). The resolution that a particular simulation contains depends upon the level of detail required for study. A theater-wide Air Model may only quantify relatively high-level parameters such as the number of planes, their types, and their locations. A low-resolution simulation might include characteristics of a particular plane in the theater such as the amount of fuel, the plane's altitude, number of weapons left, etc.

Even though computer simulations are technically challenging, they provide an excellent alternative to extensively testing hardware and training with actual systems. In fact,

computer simulations allow the examination of many strategic and tactical options in a "near-laboratory" environment; this is critical when dealing with the potential for nuclear conflict or extremely large numbers of personnel and equipment deployed on a global scale such as Operation Desert Storm(32:522).

There are several major areas in which computer simulation may play a major role in the near future. These areas include:

- *Future Weapon Systems* - the acquisition of any major weapon system such as an aircraft, tank, satellite, or ship. Computer simulation can shorten the lead time required, decrease the cost of research, development, and acquisition costs, and lessen the risk that the system may not actually counter the threat when delivered.
- *Combat Analysis* - the employment of forces in an optimal manner. Since major hostile actions do not occur very often, computer simulation offers a chance to understand and quantify the effects of various strategic and tactical options in a controlled environment. This is especially true due to political and economic problems in joint service and international operations.
- *Training* - the training of military personnel. A computer simulation can span the entire spectrum of training needs, from maintenance personnel to general officers. Typically, the purpose of a wargame is to train the wargame's players in the decision-making process.
- *Battle Management* - the technique used to provide critical information to a commander in a timely manner. Modern sensors such as those found on today's sophisticated aircraft and satellites can deluge commanders with too much information at once. Computer simulation can provide automated battle management aids and command and control systems to counteract this.
- *Manufacturing* - methods used by our country's industrial base to produce domestic products. Because of the demand for increased system performance, military systems constantly push the state-of-the-art in manufacturing technology. Computer simulation allows the exploration of process alternatives during the design stage. For

instance, metalworking processes such as forming, casting, and welding are currently on-going(32:522-523).

1.1.2 Classification of Computer Simulations.

1.1.2.1 Continuous vs Discrete. Computer simulations generally fall into two broad categories — continuous and discrete. Both methods deal with the passage of time (the independent variable) in the simulation, but they differ in how progress is measured. In continuous simulation the dependent variables of the model may change continuously over simulated time, i.e. there are no sudden "jumps".

In comparison, a discrete-event simulation (DES) model assumes the system being simulated changes state instantaneously at discrete points in simulated time. The simulation model "jumps" from one state to another upon the occurrence of an event(9:1). For example, state variables in a battlefield simulation might include aircraft position, velocity, and altitude. Typical events could include the arrival of a plane at a predetermined route point, changing course to avoid enemy contact, or the destruction of an enemy target. This research effort deals with a discrete-event simulation.

1.1.2.2 Time Driven vs Event Driven Simulation. Obviously, the simulation clock plays a key role in tracking the passage of time in both sequential and parallel discrete-event simulations. Two ways are used to determine when to advance the simulation clock — time driven and event driven. In a time driven simulation some static time increment, normally the smallest unit during which "significant" changes have occurred in the simulation, is used to advance the simulation clock. The term "significant" depends upon what is being modeled and what level of resolution is desired, i.e. it is based upon the Δt of the events of interest. If Δt is too small, then all of the significant events are tracked properly, but excessive computations are being performed. Consequently, if Δt is too large, then excessive computations are not being performed but a significant event may be erroneously omitted which impacts the rest of the simulation. For instance, if the movement of a tank on a battlefield is being modeled, then it is likely that a relatively large Δt can be used; however, if the movement of an aircraft is being modeled in that

same battlefield, then it is probable that a much smaller value for Δt will be required since the aircraft can change its location much more quickly than the tank. Regardless of what Δt is used, a time driven simulation must update the current state of all its objects each time the simulation clock is advanced.

In an event driven simulation, events indicate when it is necessary to update the simulation clock. This is particularly useful when the amount of time required between events is not well-defined, such as a battlefield simulation. It may take several seconds before an event occurs, or several events may take place virtually at the same time such as the detection of an enemy tank and the firing of a missile to destroy it. Since the type of simulation used for this research is an event driven simulation, time driven simulation will not be discussed.

1.1.3 Components of a Sequential Discrete-Event Simulation. Sequential discrete-event simulation refers to the execution of a single DES program on a sequential, i.e. non-parallel computer, and typically contains three basic data structures (9):

- Simulation clock - used to track how much progress has been made
- Next Event Queue (NEQ) - a time-ordered list of events which have been scheduled, but have not yet been executed
- State Variables - describe the various attributes of the system, collectively referred to as the "state" of the system being modeled

Each event has an associated time stamp. As the simulation clock advances, the simulation removes the smallest time-stamped event from the NEQ and executes it. This usually adds new events to the NEQ, which are always scheduled at a time greater than or equal to the current simulated time (12:745).

A DES must always select the smallest remaining time-stamped event (E_{min}) from the NEQ as the next one to be processed. If some other event ($E_{non-min}$) were selected, then it would be possible for $E_{non-min}$ to change state variables that are going to be used later by E_{min}, which is not allowed. Errors of this kind are known as causality errors

(9:2-3). Since a sequential DES executes one event at a time, there is no possibility that computations are made erroneously. Therefore no causality error can occur.

1.1.4 Components of a Parallel Discrete-Event Simulation. Parallel discrete-event simulation (PDES) refers to the execution of a single discrete-event simulation program on a parallel computer. A widely used PDES model was proposed independently by Bryant (4) in 1977 and by Chandy and Misra (5) in 1981. The system to be simulated, such as a battlefield, is partitioned into a set of components called physical processes (PPs). These PPs are usually partitioned according to an object-oriented paradigm in which each PP represents a significant object in the system. However, the simulator does not use these PPs directly. The simulation internally represents these PPs as a collection of logical processes (LPs) that communicate with each other via the sending and receiving of time-stamped messages; the time stamp denotes at what time the event is supposed to occur. LPs are designed to mimic the behavior of an existing PP. For instance, the scheduling of an event for PP_x at time t in the physical system is simulated by sending a message with timestamp t to LP_x. Although theoretically more than one LP can reside on a processor in the parallel computer at once, this research deals with only one LP per processor since BATTLESIM may be ported to an Intel i860-based Hypercube in the future which allows only one LP per node.

In a PDES, each LP has its own local simulation clock which indicates how long the LP has executed in simulation time. In effect, the single NEQ, simulation clock, and state vectors of a sequential DES are partitioned into multiple NEQs, simulation clocks, and state vectors respectively — one for each LP. Each LP's simulation clock tends to vary depending upon the computations each LP is performing. Thus, at any point in time, it is likely that all of the simulation clocks will have different values. In order to correctly simulate a PP, the corresponding LP must process incoming messages in ascending timestamp order, versus their real-time arrival order(38:29).

1.1.5 The Problem in Achieving Speedup. Model speedup is measured when there exists two implementations of the same simulation model, one that is targeted for sequential execution and one for parallel execution; it is traditionally defined as the ratio of parallel

to sequential execution times for these two different implementations (39). If someone really wanted to get a sequential DES running on an *n-processor* parallel computer, then all they would have to do is place a complete copy of the simulation in each node in the computer, assuming there was enough resources on each node such as memory. Then each node could execute its copy of the simulation at the same time as all other nodes without any fear of a processor deadlock due to a causality relationship. This simulation, generally termed a *perfectly parallel* simulation, does not achieve speedup *on a single run* since the parallel implementation is not running any faster than the sequential implementation; you just have n more copies of the simulation to analyze. However, if the simulation's objective is to create n copies of the run then *n-fold* speedup has been achieved.

If a PDES is going to achieve maximal speedup, then it must execute multiple events on multiple processors concurrently (9:3). The problem with this is that if the execution of *event a* can affect the execution of *event b*, then *event a* must be executed sequentially *before event b*, regardless of how many nodes are available for execution. The following chapter elaborates on current methods used to ensure this causality relationship is maintained. The challenge to PDES in general is to take advantage of the simulation's intrinsic parallelism by using multiple processors to execute independent events concurrently, while simultaneously executing interdependent events in sequence to maintain simulation integrity. The difficulty lies in the fact that it is often hard to know a'priori which events are independent and which are interdependent(33:1-6).

1.2 Specific Research Problem.

This thesis research involved the modification of an existing battlefield parallel discrete event-driven simulation called BATTLESIM to incorporate load balancing via spatial partitioning. Research was conducted on the Intel iPSC/2 Hypercube[1]. Several associated questions were addressed to accomplish this goal. The questions included:

1. What criteria should be used for deciding how big a battlefield sector is?

[1]The Air Force Institute of Technology School of Engineering owns a Hypercube with 8 nodes, with each node consisting of an Intel 80386 microprocessor.

2. How often should the sector size be changed?

3. What shape should the sector be?

4. How and when should battlefield players be transferred from one sector to another?

5. What kind of object should be used in the simulation to represent sectors?

6. How should battlefield scenarios be generated to test dynamic partitioning capabilities in BATTLESIM?

The premise for speedup due to spatial partitioning in a PDES is that battlefield objects physically separated by such a distance that they reside on distinct processors would not usually execute events that could affect one another; in BATTLESIM this is generally true due to the limited sensor range of battlefield objects. Therefore two objects could execute their respective events simultaneously and achieve simulation speedup. Dynamic readjustment of the spatial boundaries assigned to each processor based upon how many objects are in a sector would help improve simulation speedup further; this would alleviate scenarios in which just a few processors are handling the vast majority of the computational workload, while precious computational resources on other processors go practically unused. Since each processor would always be looking at a fraction of the total number of objects on the battlefield at any one time, each event execution on that processor should take less overall time to complete (33:4-2).

1.3 Assumptions.

1. Code and documentation from Capt Steve Soderholm, the last person to conduct research and development on BATTLESIM, is available on the Hypercube.

2. Access is provided to the Intel iPSC/2 Hypercube in the AFIT Parallel Simulation Research laboratory.

3. A 'C' language programming environment can be used on the Hypercube.

1.4 Scope.

The scope of this effort included the modification of BATTLESIM to incorporate spatial partitioning of the battlefield, the elimination of unnecessary replicated object state information, and the removal of all components supporting single-event lookahead with local rollback from the simulation. No attempt was made to make the battle simulation more realistic.

1.5 Desired End Results.

To achieve speedup which is linear in nature.

1.6 Approach/Methodology.

- Remove the components supporting single-event lookahead with local rollback from the simulation.
- Design new objects and procedures necessary for spatial partitioning.
- Remove unnecessary replicated object state information from each processor.
- Implement new objects and procedures in simulation.
- Design test scenarios for upgraded simulation.

II. Literature Review

2.1 Introduction.

There are several possible reasons to place a sequential DES on a *n-processor* parallel computer. One could be to generate a "perfectly-parallel" simulation which executes n complete copies of the simulation in the time previously required to execute just one copy of it. An example of where this might be useful includes a stochastic simulation in which statistical analysis is being performed. Since this research deals with achieving maximal model speedup with a Parallel Discrete Event Simulation (PDES) while maintaining proper causal relationships between LPs, this literature review shall focus on it exclusively.

Experienced researchers have found that the following five properties are critical deciding factors in the efficiency of a PDES:

1. **Granularity** - a factor describing how long, on the average, a processor can perform computations before it has to perform message-passing, i.e. average processer computation time per unit simulation time.
2. **High Potential Concurrency** - the inherent characteristics like short critical paths, balanced decomposition, etc. which allow a simulation to perform a significant percentage of its calculations in parallel.
3. **Distributed Geometry of Computation** - the ability of the simulation to exhibit a high degree of spatial locality (low percentage of message-passing to nearby processors) and temporal locality (events are scheduled in the near future).
4. **Balanced Process Assignment** - even distribution of the computational workload across all processors in the parallel computer.
5. **Time Synchronization Protocol** - what method is used to ensure that all processes remain "close" to each other in simulation time to ensure that the correct sequence of interdependent events is maintained (19:29).

The next few sections briefly investigate what current research has been performed recently in these areas with particular emphasis on the last three items.

2.2 The Purpose of Time Synchronization.

The purpose of time synchronization in PDES is to ensure that each LP processes its events in non-decreasing timestamp order to maintain causality constraints. Deciding what time synchronization protocol is best, or whether they are application dependent, is a topic of considerable debate in the parallel simulation research community. Two general approaches to time synchronization currently exist — conservative and optimistic, with several variants for each. Variants on the conservative time synchronization protocol are described first, beginning with deadlock avoidance.

2.3 Conservative Time Synchronization Protocols.

2.3.1 Deadlock Avoidance. As mentioned earlier, a problem occurs when a LP with an earlier local simulation time sends an event to be executed on another LP which has a later local simulation time. If the receiving LP were only allowed to execute events when their scheduled time arrives (as intended), then the receiving LP would never process the newly received event since the proper time for its execution has already passed.

The conservative time synchronization protocol, initially developed by Chandy-Misra (5) and Bryant (4) and known as the CMB protocol, prevents this situation from occurring by allowing each LP to advance it local simulation time *only when it is certain that it will receive no late-arrivals*, i.e. no event can be received with a scheduled execution time lower than the simulation time the LP is moving towards(38:29)(25:33).

Under this protocol, each LP runs its own part of the simulation and sends the results of its computations to other LPs via messages. The messages sent out by each LP are *always* in increasing order; therefore when a LP has received messages from all the other LPs, it identifies the smallest "timestamp" of those messages and calls that its new local safe time. It updates its local simulation clock to this new safe time value, and executes all events in its local next event queue (NEQ) with a time less than or equal to its local safe time. This sequence ideally continues in the simulation for each LP until the simulation finishes. BATTLESIM uses this variant of the conservative time synchronization paradigm because Soderholm considered it the most straigtforward to implement.

Unfortunately, it is possible for the processors to enter a deadlocked state in which several processors are all waiting to receive messages from each other, even if the system being modeled is deadlock free (38:29). Null messages can be used in order to avoid deadlock situations. Chandy and Misra developed a technique in which processor deadlocks are detected and corrected. This technique relies on each processor to send out null messages — messages which convey no event to be executed, but serve only to advance the receiving processor's local simulation time — to all other processors *immediately* (hence eager) when it blocks, or when it currently has no events of its own to execute(35:38). This way the receiving processors will not have to wait indefinitely for messages from other processors to update its safe time and keep on executing events (8:16). This method works as long as there are no cycles in which the collective timestamp increment of a message traversing the cycle could be zero(9:7).

2.3.2 Deadlock Detection and Recovery. Another approach developed by Chandy and Misra eliminated the use of null messages (5). This form of the conservative paradigm is similar to deadlock avoidance, but it uses a separate mechanism to detect when the entire simulation is deadlocked, and still another one to break it (9:7). Unlike deadlock avoidance, however, this mechanism does not prohibit cycles of zero timestamp increment. However, several cycles like this could seriously degrade overall simulation speedup.

2.3.3 Moving Time Windows. One way of transforming traditional PDES into a form compatible with parallel processors is to "relax" some of the causality constraints inherent in the chronological ordering of simulation events. Moving Time Windows (MTW) is based upon the premise that all events with times of occurence within a relatively small window of time may be safely considered for concurrent execution; a set of rules is used to determine which events in this set can be executed sequentially (34:34).The purpose of the window is to reduce the "distance" one must search in determining if an event with a smaller timestamp will later be received.

If the window is too small, there will be too few events available for concurrent execution because the time interval is short. If the window is too large, the simulation behaves in much the same way if no window were used at all, since other mechanisms

implicitly assume an infinitely large time window. Therefore, determining the correct window size requires application specific information (9:9).

2.3.4 Recent variants of Conservative Protocol. Message-passing between LPs accounts for a significant part of the total computational workload of a processor. In fact, the principal problem with the conservative paradigm as desribed thus far has been that the volume of null messages may greatly exceed the number of event-containing messages, thus crippling performance (35:38). Recent developments on the use of null messages to avoid deadlock are summarized below.

- *Indefinite Lazy Message Sending* - It defers sending null messages (hence lazy) until a series of null messages and an event-containing message can be combined and sent all together; this assumes that the LP sending the null messages is not deadlocked, but only idle for a period of time. Since output events on a LP are often triggered just by input events, the basic premise for this technique is that deferring the delivery of preceeding null messages to an LP is less likely to hamper its progress compared to the deferred delivery of event-containing messages. Note that a message if free to carry as many events as it wants in this algorithm, and a deadlock recovery scheme is inherent (35:39).
- *Eager Events, Lazy Null Messages* - Null output messages are stored in an output queue. Event-carrying messages, combined with any null output messages already stored and waiting to go to the same LP, are sent immediately in a composite message. When an LP requests messages, then the null message from the LP with the earliest simulation time is sent to the requesting LP (35:39).
- *Indefinite Lazy, Single Event Messages* - Unlike the algorithm described immediately above, this one saves all output messages from LPs, whether the output is an event-carrying message or a null message. Output queues can thus contain multiple event-carrying messages. Like the algorithm above, when an LP requests messages, then the null message from the LP with the earliest simulation time is sent to the requesting LP (35:39).

- *Indefinite Lazy, Multiple Event Messages* - This is the same as the algorithm directly above, with one exception. When an LP is preparing to send its composite message, it sends *all* of its stored event-carrying messages, and not just one (35:39).
- *Demand-Driven Messages* - The four algorithms just described all work by having a processor send messages on its own in simulation time order to other LPs. The same result can also be accomplished by having LPs waiting for messages request them explicitly from sending LPs via demand messages. This way messages are sent exactly when they are needed by the receiving LPs (35:39)(9:7).
- *Demand-Driven, Adaptive Messages* - This algorithm is the same as the one described immediately above, with one exception. A threshold specifying the maximum amount of stored messages is maintained for each communications path between LPs. Output of messages from an LP is stored only until the threshold for the corresponding communications path has been reached; when that threshold is exceeded, the entire contents of the output queue are sent as a single message. The threshold values change depending on the frequency of demand messages sent by requesting LPs. Since the simulation itself changes the number of messages on the communications paths dynamically, the algorithm is called adaptive (35:39).

2.3.5 Lookahead in Conservative Protocols. Lookahead refers to the ability of an LP to predict future message that it will send based upon knowledge of messages it has already received (8:17). The degree to which LPs can use lookahead can have a dramatic impact on the performance of PDES algorithms, especially conservative protocols, and can be quantified. If an LP at simulated time t can predict with complete certainty all events it will generate up to simulated time $(t + \Delta t)$, then the process has a lookahead ability of Δt (10:24). This may enable other LPs to safely process their own pending event messages.

2.4 Optimistic Time Synchronization Protocols.

2.4.1 Time Warp. There is another way to handle the problem of causality constraints. Instead of forcing the receiving processor to only go forward in time, optimistic time synchronization allows it to move backward in time. The processer moves backward

in time, a process known as *rollback*, if it receives a message from its past. When the processor does a rollback, it restores its state to a time which is less than the offending message so it can execute forward in time again (17:78).

What if the execution of this new event impacts events that have already been executed on this processor? Then those events have executed erroneously due to incorrect messages; therefore they have to be run again with the correct information, which means that the information necessary to run them again must be saved. This could involve a considerable amount of memory depending upon how far back the simulation has to go, so Time-Warp simulations use more memory in general than CMB simulations (19:34)(25:33). Therefore **the optimistic protocol can and usually does perform "incorrect" computations, while the conservative protocol can not do this because it never permits the opportunity to arise.**

Time Warp uses a technique called *fossil collection* to recover storage from messages and states that have been saved, but which are no longer necessary to store; it also utilizes *anti-messages* to cancel out the effects of messages that are suddenly erroneous due to processor rollback (16:404)(18:1). Time-Warp allows the receiving processor to keep on running events in its own next event queue without wasting precious computational time waiting for other processes to guarantee they will not send messages with earlier timestamps. If "relatively few" rollbacks have to be performed during the simulation, and if they don't rollback "too far", then a significant amount of speedup can be achieved by using this protocol.

In order to help achieve higher model speedup in a simulation, aggressive and lazy message cancellation are currently being researched as two methods within the optimistic protocol to cancel incorrect messages. These two methods are summarized next. The cancelback protocol, a method which minimizes the amount of memory needed to save previous computations in case they must be performed again due to rollback, is then shown.

2.4.2 Aggressive Message Cancellation. Aggressive message cancellation (AC) immediately cancels all messages that the simulation detects an LP has incorrectly com-

puted (11:62). This includes all messages from an LP with a time stamp greater than or equal to the rollback time. Messages are sent as anti-messages to the original recipients. This method works on the premise that most bad LP input also produces bad LP output (26:113).

2.4.3 Lazy Message Cancellation. This method is similar to AC, except that it waits until the simulation is certain that the messages sent as a result of the improper LP computation will not also be sent by the updated, correct LP computation. Therefore the time between detection of an incorrect computation and the cancellation of its messages can be very long with this method. Lazy cancellation (LC) works on the premise that a significant part of the bad LP input produces the same output as a good LP input would produce, i.e. don't cancel an output that may be right for the wrong reasons (26:113). Gafni performed tests that indicated that LC is faster and induces less traffic and fewer rollbacks than AC in most cases, but consumes more storage space (11:61).

2.4.4 Cancelback Protocol. The cancelback protocol is a variant on the optimistic time synchronization protocol designed to address Time Warp's excessive memory utilization problems. This protocol is significant because it helps ensure that Time Warp uses as little memory as is absolutely necessary. In addition to utilizing fossil collection, the cancelback protocol also uses a technique termed *cancelback* — the recovery of storage assigned to messages and states at times so far in the future that their memory would be better used for more immediate purposes (18:1).

2.5 SPECTRUM Overview.

Until 1988, no known environment existed which allowed two or more algorithms to be applied to the same application in the same environment. Therefore Reynolds and a team of researchers at the University of Virginia (UVA) set out to design a testbed that would allow for the testing of a variety of parallel simulation algorithms in a common environment (28). SPECTRUM (Simulation Protocol Evaluation on a Current Testbed using Reusable Modules) was designed to test the hypothesis that the effectiveness of parallel simulation protocols highly depends upon the applications using them, or more

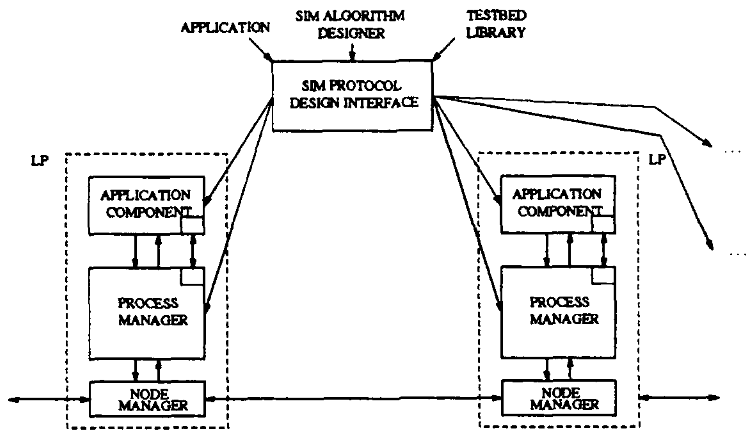

Figure 1. Block Diagram of SPECTRUM Testbed (29)

accurately, that specific *classes* of protocols work best with specific *classes* of applications (29:671). SPECTRUM supports a model of communicating processes known as logical processes (LPs), in which the original application is divided into application components which will execute concurrently. Each LP under SPECTRUM includes all the code required to form an independent process, including an application component, an LP manager, and a node manager as depicted in Figure 1 (13:1).

The LP manager provides LP-level functions to the application component, including management of an input message queue from which the application can send and receive events, initialization routines to ensure SPECTRUM is configured properly, and routines to advance the local simulation clock. The node manager provides machine-dependent functions used by the LP manager to manage passing event messages between itself and other LP managers. In addition, structures known as *filters* are used to implement time synchronization protocols. Each function in the application may have a filter which intercepts an LP-level function call and performs some protocol-specific task first (like null message generation) if desired.

Several researchers at the Air Force Institute of Technology (AFIT) have used SPECTRUM in the past as a simulation testbed, including Soderholm's version of BATTLESIM (33). Research at AFIT using SPECTRUM continues with applications in VHDL simulation (3), queuing simulation (37), computer firmware containing SPECTRUM functions (6), and of course BATTLESIM.

2.6 Dynamic Load Balancing.

In order to maximize the model speedup in a PDES, it is important that the computational workload is balanced equally among the processors. Static load balancing is an attempt to determine this ideal balance *before* the simulation actually executes. The degree static load balancing maximizes speedup is largely dependent upon how even the workload remains during the simulation — the balance may be fine for the overall simulation but may still be terrible at particular points in the simulation. *Dynamic load balancing (DLB)* initially distributes the workload just like static load balancing, but it then looks for ways to shift the workload between processors while the simulation is running (12:746). The workload may be distributed across processors on the basis of time (temporal locality), space (spatial locality), or some other common characteristic. For example, Reiher investigated ways to exploit temporal locality in his paper dealing with dynamic load management in the Time Warp Operating System (27), and Nicol explored ways to dynamically partition the domain of a battlefield simulation (24). An even workload balance is desired to ensure that if objects cluster on the battlefield, then no processors are carrying an inordinately large portion of the workload while others carry a relatively small portion, deceasing overall speedup due to inefficient use of system resources. This research effort utilizes spatial partitioning in preparation for dynamically changing sector size in future BATTLESIM research.

2.7 Boundary Crossing Events on a Battlefield.

In 1989, Frederick Wieland and other researchers developed a ground combat simulation to experiment with data partitioning in a distributed computing environment (41). They wanted to derive an effective "proximity detection" algorithm for objects moving in space to detect each other, i.e. determine at what simulation time two objects are within sensor range of each other when their state information — such as position and velocity — is located on different processing nodes.

They devised a scheme in which the battlefield was divided into rectangular regions called *grids*, with each one responsible for a subset of the simulation's total state informa-

tion. In order to correctly handle one object sensing another object in another grid, they decided some sort of data replication was necessary.

The proximity detection algorithm they created consisted of two parts: an object detecting another object within the same grid (unit-unit proximity detection), and an object detecting another object when it resides in another grid (grid-grid proximity detection). In the first case, the grid solved a quadratic equation which calculated when an object would enter and leave another object's sensor range. In the second case, three events were added: 'Add Unit' to add an object to another grid when its sensor range entered the new grid, 'Change Grid' when the object's center of mass crossed from one grid to another, and 'Delete Unit' to remove an object from a grid when its sensor range left that grid (41).

In their analysis, Wieland and the others concluded that this algorithm did not impose any synchronization constraints between grids, allowing each one to process asynchronously. The algorithm also allowed each object to possess a different sensor range which could change dynamically.

III. BATTLESIM Simulation Model

3.1 Introduction.

This chapter gives a high-level description of the BATTLESIM simulation model from its foundations two years ago until today. First Section 3.2 outlines the latest requirements for BATTLESIM. Section 3.3 provides a system overview describing what BATTLESIM is and how it works. The design philosophy used to make changes to BATTLESIM is discussed in Section 3.4, including maximizing the reuse of existing code and using an object-based design approach. The benefits of using an object-based design are discussed. These benefits include the ability to change data structures easily and the ability to use a "hierarchical" approach. Finally, several design goals are presented in Section 3.5. Four of the subsections are making parallelism as transparent as possible (Section 3.5.1), making the time synchronization protocol used as transparent as possible (Section 3.5.2), making software integration of the various packages transparent (Section 3.5.3), and balancing program efficiency and clarity (Section 3.5.4). The other three include supporting multi-processor scaling (Section 3.5.5), maintaining simulation integrity (Section 3.5.6), and making it easy to change configurations (Section 3.5.7).

3.2 Current Requirements.

While the main thrust of this phase of research with BATTLESIM was to implement spatial partitioning, several additional requirements were also added. The extra requirements include:

- Upgrade the current version of TCHSIM encapsulated within BATTLESIM, while establishing "hooks" in BATTLESIM which allow future upgrades to be accomplished in a easy, modular fashion.
- Apply principles of software engineering — like data abstraction, information hiding, modularity, localization, and software reuse — to any new code developed for BATTLESIM to support future maintenance and understanding.
- Provide the ability to have multiple input scenario files (1 per LP), if desired.

- Make it easy to fully or partially replicate any state information through the use of a message generation package.
- Make it easy to change the partition size of a battlefield sector.
- Ensure that lower-level objects have no knowledge of the battlefield environment in which they run through the use of an object-based approach.
- Eliminate unnecessary data replication within sectors.
- Allow small frequent changes to a given scenario to be made in such a way that recompilation of the application is not required, e.g. using command-line arguments.
- Provide the capability for BATTLESIM to run sequentially on a parallel or sequential platform, and in parallel on a parallel platform.
- Give scenarios the ability to support multiple sectors per logical process (LP).
- Allow objects to have non-sequential integer identifiers, and allow more than 1024 objects total in a given scenario.
- Allow BATTLESIM to handle the case where there are no objects in a sector and there are no objects owned by an LP.
- Have BATTLESIM provide an object with sensing abilities when it is not moving and still be considered valid if it has route points left.
- Incorporate the ability for collisions to occur between objects without their centers of mass intersecting.
- Ensure a scenario executes properly with negative sector boundaries and object route points.
- Generate a method to handle objects exiting the battlefield so the scenario can continue executing, e.g. "wrap-around".
- Allow objects with no sensing capabilites (zero sensor range) to move about properly on the partitioned battlefield.

These requirements for the present version of BATTLESIM were indeed accomplished; the planning and implementation to meet these requirements is discussed at length in upcoming chapters.

3.3 System Overview.

BATTLESIM is designed to simulate objects (missiles, aircraft, and land vehicles) moving along predetermined route points in empty space until they either sense another object or a battlefield sector boundary. If an object senses another object, then it reacts by either attacking, evading, or continuing along its planned path of movement; if it senses a sector boundary, then it determines what kind of boundary-crossing event is taking place and executes it. Boundary-crossing events involve either object replication into a sector because the object has visibility there, object removal from a sector because it no longer has visibility there, or updating object ownership because the object physically resides in a new sector.

All objects in a given scenario, along with their associated attributes, are specified in one or more scenario input files retrieved by BATTLESIM at run time. These objects are dynamically destroyed when they reach their last route point. Missiles are dynamically created during the course of the simulation, and are the only means of attack available since no other weapons are provided.

Figure 2 illustrates how the simulation currently executes. First of all, the user must generate scenario and map files (both described in Chapter 6) which describe the objects in the simulation and the sector-to-LP mappings, respectively. Note that *each logical process (LP) which is part of the scenario must have access to a scenario file during initialization, regardless of whether it is designed for use by only that LP or by multiple LPs.* The mapping file defines the sector-to-LP mappings for the scenario. After these two files have been created and the executable BATTLESIM code generated, the user executes the simulation by providing the application name and command-line arguments. If specified through the use of two conditional compilation 'define' variables, BATTLESIM generates scenario status information which it sends to a graphics display file as well as the screen. While the screen output is only suitable for reading, the graphics display file

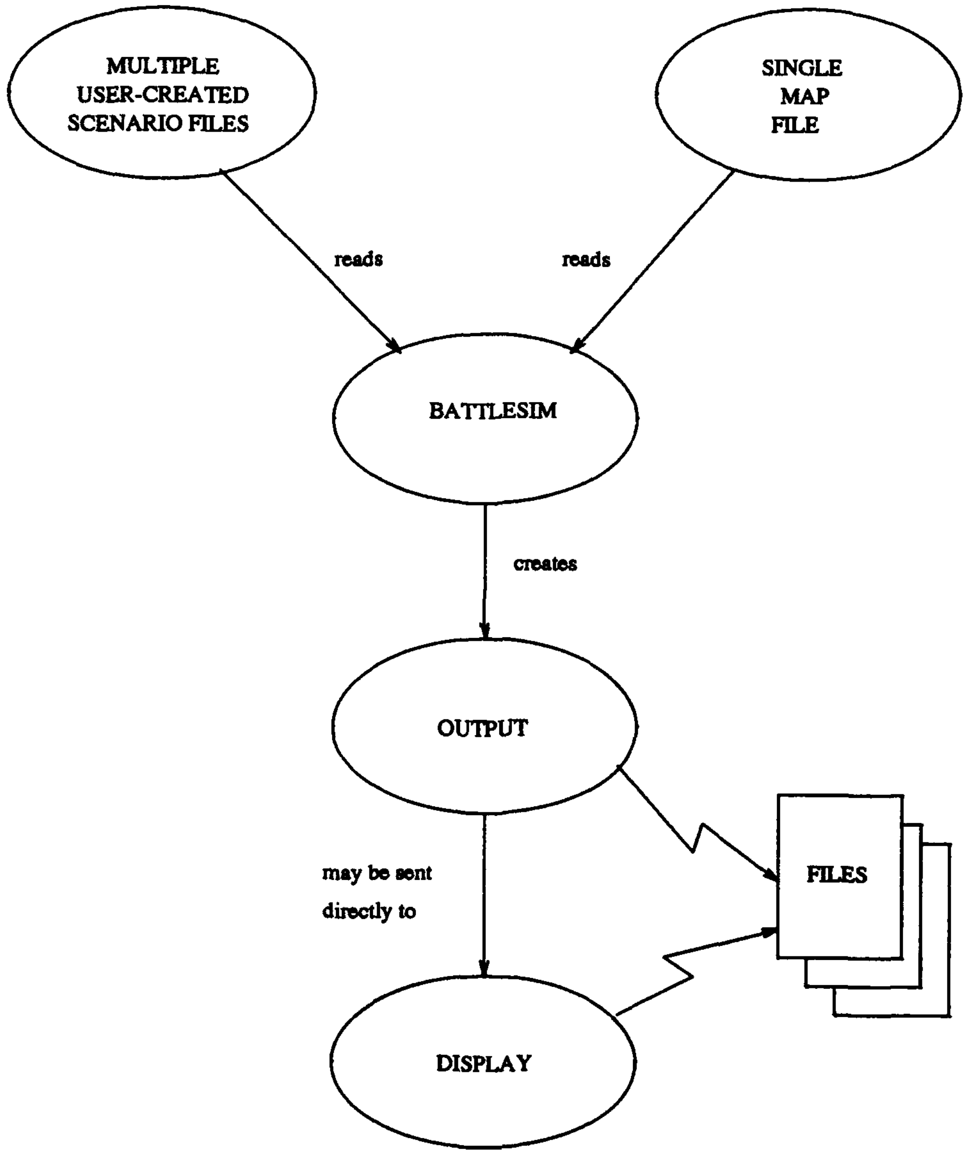

Figure 2. BATTLESIM "Big Picture"

is in a special format designed to be read by a display driver which depicts the battlefield scenario.

3.4 Design Philosophy.

In the process of incorporating support for spatial partitioning within BATTLESIM, a specific design philosophy was followed. That philosophy, which is purposely designed to follow the one previously outlined for parallel simulation design efforts at the Air Force Institute of Technology (14), includes the factors discussed in the following sections.

3.4.1 Maximize Reuse of Existing Code. Over the last two years since development of a standard Air Force Institute of Technology battefield simulator began, significant effort has been expended in the creation of code which can be used as the cornerstone for further research into parallel battlefield simulation. Therefore, if software components existed which could be used directly (or upgraded with minimum effort) in BATTLESIM, those components were 're-used' instead of developing new code. Code which falls into this category includes:

- **TCHSIM** - a C-based parallel discrete-event simulation environment designed by Dr. Thomas Hartrum to allow experimentation with various application models, including BATTLESIM, without the need to re-implement the "basics" every time (15:1).
- **Generic Linked List Package** - a package of routines designed by Rizza to support instantiation, modification, and deletion of three types of queues implemented as linked list structures.
- **TCHMAP** - A package of routines which allow the mapping from any specified object to a logical process (LP). This is useful for mapping sectors in the battlefield to a particular LP (15:14).
- **RIZSIM** - In preparing to build upon RIZSIM for this research effort, some sections of code were considered detrimental to implementation of spatial partitioning within BATTLESIM and were thus removed. These sections of code are described in detail in Chapter 6.

3.4.2 Utilize an Object-Based Design. Booch differentiates between an object-oriented and object-based design; he states that an *object-oriented* design supports polymorphism and inheritance, while an *object-based* design does not (14:3). BATTLESIM cannot be called object-oriented since it does not support these qualities due to the inherent limitations of the 'C' programming language. However it is certainly possible — and indeed desirable — to make BATTLESIM object-based by encapsulating objects along with their methods into abstract data types (ADTs).

There are several software-engineering related benefits for BATTLESIM because of the use of an object-based design in new code development. Two of them are the ability to change data structures easily due to information hiding, and the ability to use a hierarchical approach to support different levels of data abstraction (see Appendix B for others). For example, in BATTLESIM:

- the `battlefield` is composed of `sectors`
- a `sector` object is composed of a `playerset` and other attributes
- a `playerset` object is composed of `players`
- lastly, a `player` object is composed of several `low-level fields`

Each of these four kinds of objects exist at a different level of abstraction. The advantage of using data abstraction in BATTLESIM is that it does not force the user to learn the entire application in order to conceptually understand how it works. Then, as the user desires a more in-depth understanding, he can investigate progressively lower levels of abstraction.

3.5 Design Goals.

3.5.1 Make Parallelism Transparent. One of the goals of BATTLESIM is to make the steps required to run it in parallel as transparent as possible. Currently, the only parallel processing platform that BATTLESIM supports is the Intel iPSC/2 Hypercube. Since BATTLESIM is based upon the concept of communicating logical processes (LPs),

the application *must* be partitioned into LPs before it can be executed in parallel. There are two different options for accomplishing this:

1. **Option 1** - Design the BATTLESIM simulation model as a single LP, and then break it into several LPs.
2. **Option 2** - Design the BATTLESIM simulation model as a collection of LPs, and then map these to the Hypercube for execution (14:4).

Since the latter option is what BATTLESIM employs, the user *must* understand that the Hypercube is composed of LPs, and at the application level must generate map files which identify which sectors he wants to be controlled by which LP.

3.5.2 Make Time Synchronization Protocol Transparent. Another long-range goal of BATTLESIM is to eventually investigate the effects on overall simulation speedup due to the use of different time synchronization protocols when executing a parallel simulation, making the use of different protocols as transparent as possible. AFIT has been experimenting with the SPECTRUM simulation testbed developed by UVA (14:1). BATTLESIM, like it predecessor RIZSIM, employs the conservative time synchronization protocol developed by Chandy and Misra (5). SPECTRUM can be used to provide this, or any other time synchronization protocol. To do so, a *filter* would be devised which describes the protocol to SPECTRUM. This approach would theoretically simplify experimentation using various time synchronization protocols with an application while minimizing the changes required to the application itself. However, experience suggests that it is not as simple as this.

3.5.3 Make Software Integration Transparent. There are three major software packages which are integrated into the BATTLESIM simulator: the BATTLESIM application code, the SPECTRUM testbed, and the TCHSIM generic simulation driver. Some of the functionality contained in each of the three packages sometimes overlaps. Soderholm unfortunately meshed all three packages together in such a way that a *significant* amount of effort was necessary to use to new versions of SPECTRUM and TCHSIM. Therefore the old versions of these two packages had to be "pulled out" of the existing BATTLESIM code,

and then the new versions added in such a way that future upgrades could be accomplished in an object-based manner.

3.5.4 Balance Program Efficiency and Clarity. While this is more of a subjective factor, it is undoubtedly an important one. Programmers usually face a dichotomy — they can either make a program extremely efficient, possibly making it difficult to comprehend and maintain, or they can make it simplistic at the potential expense of execution efficiency. BATTLESIM was designed to be used for research in an academic environment, with future modifications an inevitability. Therefore program clarity had to be a consideration. However, when more advanced algorithms and data structures could be applied without significantly impacting the application's inherent maintainability and understandability, updates containing these changes were made.

To see how this factor affected the decisions on new BATTLESIM data structures, the reader should refer to Chapter 6 where implementation is discussed.

3.5.5 Support Multi-Processor Scaling. As mentioned earlier, battlefield simulations are often run to simulate large areas of the world like Southwest Asia. The larger the battlefield is in a given scenario, the more desirable it is to use a computer with more processors in order to maintain reasonable simulation execution times. Since the current trend in state-of-the-art parallel computing platforms is to increase the number of processors as well as memory and potential connection paths, it is imperative that BATTLESIM be able to execute properly on computers with more than eight processors.

3.5.6 Maintain Simulation Integrity. Simulation integrity in a deterministic simulator is the ability of the simulation to obtain the same results every time it is run if the same input scenario is used, regardless of how many nodes the scenario requires. BATTLESIM simulation integrity is verified by performing two steps: checking the graphics output file generated with one processor to be sure it is correct, and then comparing graphics output files generated by scenarios using mutiple processors with that obtained from just one.

3.5.7 Make it Easy to Change Configurations.

3.5.7.1 Specifying the Configuration. There are several pieces of information the user must specify to BATTLESIM in order for it to run properly. This information includes:

- The hardware platform upon which BATTLESIM is supposed to execute. The hardware platforms BATTLESIM currently supports include the following.
  - the Intel iPSC/2 Hypercube.
  - the Sun SparcStation 2.
- The number of LPs.
- The number of physical processors (PPs) available on the hardware platform.
- Assignment of LPs to processors.
- The dimensions of the battlefield.
- The players involved in a given simulation scenario, along with their corresponding attributes.

In order to run BATTLESIM on different processing platforms, different executable versions of BATTLESIM must be generated (one for each). Both the SparcStation and the Hypercube have unique system requirements and capabilities which differentiate it from the other. The SparcStation version includes:

- utilizes interface0 (sequential version) of TCHSIM
- alway runs sequentially as LP0
- does not use any SPECTRUM function calls
- has SparcStation-specific interface routines
- always utilizes a single scenario input file containing all objects

The Hypercube version includes:

- utilizes interface0 or interface1 (parallel version) of TCHSIM
- can run either sequentially on the host as LP0 or in parallel on 1-8 nodes
- does use SPECTRUM function calls
- has Hypercube-specific interface routines
- utilizes either single or multiple input scenario files

Configuration information is retrieved from one of three places — scenario files, map files, and command-line arguments — which are specified by the user at run time. This information can be changed *without recompilation or relinking.* This makes experimentation with various configurations as simple as possible.

3.5.7.2 Specifying How to Partition Data. During intialization each LP *should* instantiate only those players it has been assigned from its scenario file. Therefore some sort of data partitioning paradigm would be useful. There are four ways of handling this:

- **Approach 1** - The LP could first read a map file describing the player to LP assignments. Then the LP could read a scenario file describing *ALL* the players in a given simulation, and instantiate *only* those players it has been assigned. This avoids the need to preprocess the data, but it requires the LP initialization code to be aware of the LPs and parallelism.
- **Approach 2** - The LP could first read the same scenario file describing all the players in a given simulation. It could then read the same map file consisting of player-to-LP assignments, and *delete* what it does not need.
- **Approach 3** - The LP could first read the same scenario file describing all the players in a given simulation. It could then read the same map file consisting of player-to-LP assignments, and *ignore* what it does not need.
- **Approach 4** - The single scenario file described above could be divided a'priori into separate scenario files for each LP. Then each node would not need a separate mapping file describing which players it is supposed to instantiate, but would instead instantiate copies of all the players described in its scenario file. This requires extra

work before executing the simulation in dividing up the players into the appropriate LP scenario file, but would provide the desired effect of making the partitioning scheme transparent to BATTLESIM(14:5).

The current parallel version of BATTLESIM takes the fourth approach — it requires the user to manually separate information on each player into separate scenario files, with each LP reading the contents of *one, and only one* scenario file during initialization. This approach is pragmatic, since each node should only be aware of players for which it needs to perform some computations or pass messages.

IV. Parallel Considerations in Spatial Partitioning

4.1 Introduction.

In his thesis, Soderholm discussed at length the combination of elements in the conservative and optimistic time synchronization protocols — a method he termed *a hybrid approach.* While he discussed such topics as object partitioning among LPs, data structure requirements, and the pros and cons of spatial partitioning, he did not fully elaborate on what parallel considerations must be taken into account when implementing spatial partitioning in BATTLESIM. This chapter addresses those considerations. Section 4.2 discusses new ways of accomplishing object partitioning. Section 4.3 reviews the definition for an LP, while Section 4.4 discusses old and new functional requirements and what factors have affected them. Section 4.5 outlines the events in the last version of BATTLESIM, and discusses three new types of events designed to handle sector boundary-crossing. Lastly, Section 4.6 explains some of the complications which must be addressed in spatial partitioning.

4.2 Object Partitioning.

One of the greatest opportunities for BATTLESIM speedup lies in the reduction of the number of objects a given LP must check before determining its next event for a given object. Since the number of objects a given LP must check depends upon how many exist in its data structures, a reasonable approach would be to reduce the number of objects in those data structures — regardless of what kind are utilized — to an absolute minimum.

The "traditional" approach to partition data in a battlefield simulation has been to spatially partition the battlefield into several smaller, non-overlapping units known as *sectors.* In a spatially partitioned parallel battlefield simulation, each LP is responsible for one or more sectors in the battlefield, and is only responsible for tracking and making computations for objects which are contained within its sectors. Therefore, if two objects are far enough apart that they reside on different LPs, then they generally should not perform any operations which would impact each other since each LP is not aware of the

others' objects. This in turn should allow the objects to execute their events simultaneously and achieve simulation speedup.

The reason for this is not particularly difficult to understand. Since the number of objects an LP must check has been reduced, the time required to determine *what* the next event is for that object is reduced. This is turn results in a lower event execution time since checking sensors is an integral part of every event execution. However, there is a potential problem with the idyllic "traditional" approach just outlined. Section 4.6 contains details on the complication and techniques for handling it.

It is interesting to note that sectors can conceptually be any size or shape as long as they are no larger than the battlefield itself. Commonly used sector shapes include hexagons, rectangular cubes, squares, and strips (see Figure 3). For example, previous research by Moser used strips to spatially partition the area in which pool balls were traveling (22). However, a requirement for this research was that objects could easily cross sector boundaries in all three spatial dimensions (x, y, and z in the cartesian coordinate system) if desired. Most of these structures have inherent limitations which limit their usefulness in spatially partitioning a battlefield. Those limitations are:

- **SQUARE** - This would work, but it unnecessarily limits the flexibility a sector shape may acquire when sector boundaries change.
- **STRIP** - This would also work, but it unnecessarily limits future enhancements to BATTLESIM by allowing objects to cross sectors in just two spatial dimensions instead of all three.

A rectangular cube combines the best qualities of the other structures while avoiding their inherent limitations. It is simple enough to formulate straightforward mathematical equations for boundary crossings, and yet allows considerable flexibility in how a sector may dynamically change size (it could change in 1, 2, or all 3 dimensions at once if desired). ***Therefore a rectangular cube which extends to the top and bottom of the battefield in the Z axis is used as the sector shape for this research.***

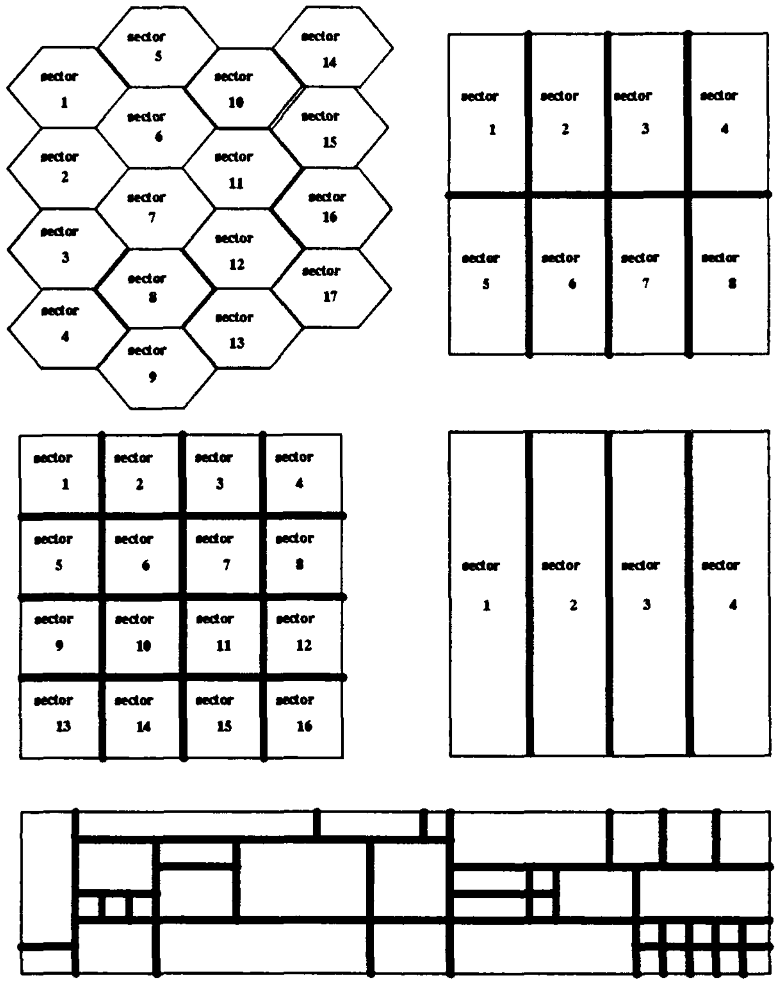

Figure 3. Examples of How to Partition a Battlefield

4.3 Definition of a Logical Process.

BATTLESIM was originally envisioned to run on the Hypercube using the SPECTRUM simulation testbed developed by UVA. SPECTRUM itself is a simulation support environment for communicating LPs, which are *not* necessarily the same as physical processes (PPs). In fact, one or more LPs can reside on the same PP (node) in the Hypercube. BATTLESIM utilizes one LP per node because it allows for the capability to port BATTLESIM to the Intel iPSC/860 in the future, since the iPSC/860 can only handle one task (LP) per node.

4.4 Functionality Requirements.

4.4.1 Soderholm's Approach. The previous version of BATTLESIM contained two primary data structures, a next event queue (NEQ) and a container structure called the "master object array". The NEQ contained all the future events for a given LP, while a master object array existed on each LP to contain the players present in the simulation.

4.4.2 Current Approach. If maximum speedup is going to be obtained from a spatially partitioned PDES like BATTLESIM, then all processors must be equally loaded throughout the simulation. The issue of workload distribution must be solved prior to selecting an algorithm for spatial paritioning. At least three approaches are possible:

- Dynamically change the battlefield's sector to LP mapping assignment.
- Dynamically change the size of the battlefield sectors.
- Dynamically change the number of sectors in the battlefield.

While all three are technically feasible, considerable interest in the second approach exists in the research community as evidenced by several published articles and conference proceedings (40)(24) (23); therefore this approach was chosen. The *new* structures required to partition the battlefield into sectors include:

1. a `player` object - a representation of a physical battlefield entity. It consists of state variables and methods to access them.
2. a `playerset` object - a container object used to manage storage of all `player` objects either residing in or possessing visibility into a sector. It consists of state variables and methods to access them.
3. a `sector` object - a physical portion of the battlefield. It contains a handle to its own playerset, as well as state variables and methods to access them.

Each LP in the simulation manages events associated with one or more battlefield sectors through its own NEQ. Since a NEQ may contain sensor check events generated on behalf of objects residing in other LP's sectors, *state information in each event must contain information specifying which sector and player the event is associated with.*

4.5 *Event Requirements.*

4.5.1 Soderholm's Approach. Soderholm defined a total of six event types for BATTLESIM (33:3-11). Those events include:

- **Reached Turnpoint** - This event indicates that an object's next event is arrival at its next route point. It is important to realize that *this is the initial event scheduled for all objects during initialization.* The simulation is over for an object when it has no more route points. If no other events are scheduled, a reached turnpoint event is scheduled provided there are route points remaining.
- **Entered Sensor Range** - Each object has zero or more sensors. This event determines if a given object will enter the sensor range of any *other* objects in the battlefield between its current and next route points. If so, then an 'Entered Sensor Range' event is scheduled at the time when this will occur, and an intermediate route point is added at the appropriate location.
- **Made Sensor Contact** - If an object determines that *any* other battlefield objects will enter its sensor range, then it schedules a 'Made Sensor Contact' event and adds an intermediate route point at the appropriate location.
- **Ordnance Released** - This event is scheduled when an object attacks another one. If it does, then a missile is dynamically created. The missile is given three route points: the current location of the object releasing the missile, the current location of the target, and the target's next route point. If the missile reaches its final route point, it is considered to have missed the target and is destroyed.
- **Ordnance Reached Target** - This event is scheduled when a missile succeeds in reaching the target before flying all of its assigned route points. The missile and its target are destroyed since a hit always results in complete destruction of the target.
- **Collision Distance Reached** - This event is scheduled when two objects physically come in contact with each other.

4.5.2 Current Approach. All of the event types defined previously are kept. In addition, other event types are required to let the simulation know when it is appropriate

to copy a player from one sector into another one it is entering, when to change a player's owning sector, and when to remove an object from a sector it is leaving. An approach similar to that described by Wieland is implemented (40). The necessary events to support spatial partitioning in BATTLESIM include:

- **FRONT_END_SENSOR (FES)** - indicates that the 'leading edge' of an object's maximum range sensor is entering another sector.
- **CENTER_OF_MASS (COM)** - indicates that the physical location of the object, i.e. its center of mass, is entering the gaining sector. This can only take place *after* the object has executed a FRONT_END_SENSOR event, if it has any sensors with a non-zero range.
- **BACK_END_SENSOR (BES)** - indicates that the 'trailing edge' of an object's maximum range sensor is leaving a sector. This can only occur *after* an object's center of mass has crossed into the gaining sector.

One of the requirements for the current version of BATTLESIM includes the ability to support objects with no sensors (equivalent to a sensor range of zero). In order to do this, the COM event has to include the capability to mimic the functionality within the FES and BES events to ensure proper BATTLESIM execution. Details of how this, as well as other changes to BATTLESIM, are described in Chapter 6.

4.6 Event-handling Complications

A potential problem exists with handling events that have been computed locally but should execute on a different LP. If an event exists in an LP's NEQ merely because an object can sense into a sector owned by that LP, then that event *must* be sent back to the LP containing the object. This is necessary because objects are responsible for determining their own next event based upon potential events in their current sector, and sectors in which they have visibility. In other words, ***each LP in BATTLESIM should only execute events for objects it "owns", i.e. that physically reside within one of its sectors.*** This information dependency could potentially force the parallel simulation into a sequential

processing mode if there is a strong dependence between sectors, seriously crippling overall performance.

If an object has a large sensor range which extends into sectors other than the one in which it physically 'resides', then that object must somehow have visibility into those sectors to correctly determine its next event. Three ways exist for generating object visibility into another sector: full, partial, and no object replication.

The first method is full object replication in which every LP has full knowledge of every object in the scenario — the method used by Soderholm's version of BATTLESIM. The advantage to using this method is that it is conceptually the easiest to understand and the most straightforward to implement. All LPs can use the same scenario file containing all the scenario's objects. Whenever a sensor check for an object is required in another sector, then the object can request a sensor check event be computed there. The event associated with that sensor check can be sent back to the LP containing the object. However, this technique requires that *object information be obtained and updated on every LP whether it is ever used there or not.* This could impose a significant and unnecessary performance penalty.

The second method is partial replication. It involves replicating the entire object in all sectors that it has visibility, having each replicated object perform a sensor check, and letting the sectors pass back the sensor check events to the requesting object. This method's advantage is that it eliminates the creation of replicated objects that will never be utilized — replicated objects are created only when they are needed, and exist only as long as they are being used to generate sensor check events. This technique requires the generation of one kind of new message type not already in BATTLESIM — one to hold the entire contents of an object. The disadvantage of this technique is that it is harder to implement than full replication. It still requires all replicated objects to be updated every time the original object is; it also forces the retention of object state information which is unnecessary to compute an object's sensor check event.

The third method is no object replication. Instead of maintaining replicated objects on each LP, an object requests that sectors in which it has visibility generate sensor check

events based upon the object's position and sensor range into each sector — the only information necessary. The resulting sensor check event could then be passed back to the requesting object. This method's advantage is that it reduces the size of messages specifically requesting a sensor check for an object in another sector, and it eliminates the need to update replicated objects. However this requires the generation of two more message types; one for the object to move from one sector to another, and another one to pass the object's position and sensor range to the sector in which it has visibility.

V. *Discussion of New Partitioning Algorithms*

5.1 *Introduction.*

New boundary-crossing algorithms are necessary to support the movement of players from one sector to another in BATTLESIM's partitioned battlefield. Section 5.2 covers in more detail why the new algorithms are indeed required. Section 5.3 describes the general equation used to determine when a player moves from one sector to another. Sections 5.4, 5.5 and 5.6 explain what refinements are made to the general boundary-crossing equation for the front end sensor (FES), center of mass (COM), and back end sensor (BES) events respectively. Section 5.7 explains how these algorithms are used collectively to determine what the next potential boundary-crossing event for a player is in a particular sector.

5.2 *Why New Algorithms Are Required.*

As mentioned previously, BATTLESIM contains three new event types to support spatially partitioning the battlefield — FES, COM, and BES. The previous version of BATTLESIM contained algorithms which indicated when a player's sensor range or center of mass intersected another player; unfortunately, these algorithms did not support the determination of when a player's maximum sensor range or center of mass would intersect a specified plane that is perpendicular to the x, y, or z coordinate axis. The use of a rectangular cube as the sector shape causes the need to find player intersections with the planes. Thus a general algorithm which determines the time required to intersect that plane from a specified point in the battlefield area is necessary.

While the battlefield is obviously three dimensional, the general algorithm as it is described in the following section is designed to be used on each dimension individually and return three distinct intersection times — one for the plane that is perpendicular to the x coordinate axis, one for the y, and one for the z. Therefore, while the explanation of this general algorithm as it applies to the three boundary crossing events is provided in terms of the x coordinate axis, **the same argument can be readily applied to the other two battlefield dimensions.**

5.3 General Boundary Crossing Equation.

In order to develop an equation to determine when in the future a specified player will intersect a given sector boundary (plane), two quantities must be known: the position of the player as a function of time, and the position of the sector boundary as a function of time. In BATTLESIM, the position may be based upon the player's center of mass or its maximum sensor range.

The player position used depends upon which type of boundary crossing event is being computed. If it is a COM event, then the position used is the player's center of mass; if it is a FES or BES event, then it is the player's center of mass plus or minus its maximum sensor range, respectively. The boundary-crossing event definitions used to describe the player's position include:

- **FRONT END SENSOR** - that sensor on a player which has the ability to sense the furthest *in front of* the player's path of movement.
- **CENTER OF MASS** - that location within a player's physical dimensions at which the player's mass can be considered to be concentrated.
- **BACK END SENSOR** - that sensor on a player which has the ability to sense the furthest *behind* the player's path of movement.

Notice that although there could be several sensors for a player, the one used exclusively in the general equation will be *the one sensor which meets the definitions for FES and BES above.* Since all sensors have a cylindrical range (circular in the x and y coordinate axes), the sensor meeting the definition for FES will automatically meet the definition for BES, and vice versa.

Two new variables are defined for the general boundary-crossing equation. These variables are:

- $x_{present}$ - The present location of an player in the x coordinate axis. In the equation, this is the *starting point* used.

- x_{future} - The future location of that same player in the x coordinate axis. In the equation, this is the *sector boundary* to be intersected.

Figure 4 is one of several diagrams in this chapter which depicts all potential starting and destination points in the general equation. In figure 4, a FES event with a positive x velocity component has its starting point $x_{present}$ located at $x_1 + max\_sensor\_range$ because the starting point of interest is *the "location" of the player's front end sensor*, i.e. the furthest point in front of the player from which sensor information can be obtained. A COM event in the same diagram uses a starting point $x_{present}$ of the center of the player itself (each player has a non-zero radius), which is located at x_1 regardless of the x velocity component. A BES event in figure 4 with a positive x velocity component uses a starting point of $x_1 - max\_sensor\_range$, which is *the "location" of the player's back end sensor*, i.e. the furthest point in back of the player from which sensor information can be obtained. Lastly, the value of x_{future} in the same diagram is located at the maximum x sector boundary. With this information in mind, the derivation of the general boundary crossing equation can begin.

The boundary crossing equation describing the current position of a player in the x coordinate axis as a function of time which supports the ability to use non-zero acceleration is:

$$x(t) = x_1 + v_{x1}t + \frac{1}{2}a_{x1}t^2 \tag{1}$$

where

x_1 = initial position of player

v_{x1} = initial velocity of player

a_{x1} = initial acceleration of player

The plane the player is approaching is in fact the next sector boundary the player should cross (in the x coordinate axis) if it continues on its current path of movement.

The position of the sector boundary in the x coordinate axis as a function of time is an equation consisting of one of two values:

$$x(t) = x_{min} \qquad \text{or} \qquad x(t) = x_{max} \tag{2}$$

The value used depends upon whether the x velocity component is positive or negative, and what kind of boundary event is being checked. For example, if the FES event for a player with a positive x velocity component is being computed, then the value x_{max} is used; if the x velocity component is negative, then the value x_{min} is used. Even though Equation 2 is specified as a function of time, the position of the sector boundary in the x coordinate axis is in fact independent of time as far as the boundary-crossing equation is concerned.

Two assumptions must be made in order to integrate Equations 1 and 2 into the general boundary crossing equation. These include:

1. Acceleration of the player in question remains constant from its current position to intersection with the sector boundary.
2. The plane in which the player is moving is perpendicular to the x coordinate axis.

The first assumption is valid since the BATTLESIM model uses a constant velocity for all of its players. The second assumption is likewise valid since all sector boundaries are defined to be perpendicular or parallel to the three coordinate axes.

The general form of the boundary crossing equation is created by setting the present location of the player $x_{present}$ equal to the expected future location of the player x_{future}. These quantities are represented by the Equations 1 and 2, respectively.

$$x_{present} = x_{future} \qquad \Longrightarrow \qquad x_{present} - x_{future} = 0 \tag{3}$$

Note that the general boundary crossing equation is a quadratic equation of the general form:

$$at^2 + bt + c = 0 \tag{4}$$

where

a = acceleration of player

b = initial velocity of player

c = constant dependent upon boundary event type

When the acceleration for a player has a value of zero, Equation 3 can be solved for time t:

$$x_{future} = x_{present} + v_{x1}t \quad \Longrightarrow \quad t = \frac{x_{future} - x_{present}}{v_{x1}} \tag{5}$$

Since the BATTLESIM currently uses acceleration values of zero for its players, Equation 5 is the general boundary crossing equation. The values that x_{future} and $x_{present}$ acquire in Equation 5 are in fact unique to every boundary crossing event, and will be described in the upcoming boundary event sections.

5.4 Front End Sensor Algorithm.

For a FES event, the value used for x_{future} in Equation 5 when the x velocity component is positive is x_{max} – the sector's maximum x coordinate — because x_{max} is the next sector boundary that the player's front end sensor will cross. Likewise, the value x_{min} — the sector's minimum x coordinate — is used when the x velocity component is negative, since x_{min} is the next boundary the player's front end sensor will cross in that situation.

To determine when the maximum range front end sensor of a player will cross the appropriate sector boundary, Equations 1 and 2 are set equal to one another as shown in Equation 5. This corresponds to when the position of the front end sensor equals the position of the sector boundary in which the front end sensor currently resides. The roots of the resulting equation are then obtained like any other quadratic equation.

Knowing *when* to use the results of the boundary crossing equation is just as important as *how* to derive them. A valid front end sensor event is one which has not yet occurred for a player in a given sector, and which is supposed to occur. A front end sensor event should *not* occur when:

- the player has a maximum sensor range of zero
- the FES event for the player has already occurred in the given sector

The first requirement is determined easily enough by a function which returns the player's maximum sensor range from all existing sensors. The second requirement takes more investigation. In order to ensure a valid sensor event *for any of the three boundary crossing events* has not already occurred in the x coordinate axis, one of two conditions must be true:

Condition 1: $x_{present} < x_{future}$ (with $+v_{x1}$)

Condition 2: $x_{present} > x_{future}$ (with $-v_{x1}$)

Figure 4 is an example of where the conditions defined in condition one are true, making this a valid front end sensor event **for sector three**. In the diagram, player one is traveling through sector three towards sector four, and player one's front end sensor has *not yet reached* the position x_{future}. Since this is a FES event with a positive x velocity component, the location of the player's maximum range front end sensor in sector three is:

$$x_{present} = x_1 + \text{max\_sensor\_range}$$

The location of the sector boundary the front end sensor should be approaching for sector three in figure 4 is:

$$x_{future} = \text{sector's maximum x coordinate} = \text{max\_x}$$

The use of these two values in Equation 5 results in the expression:

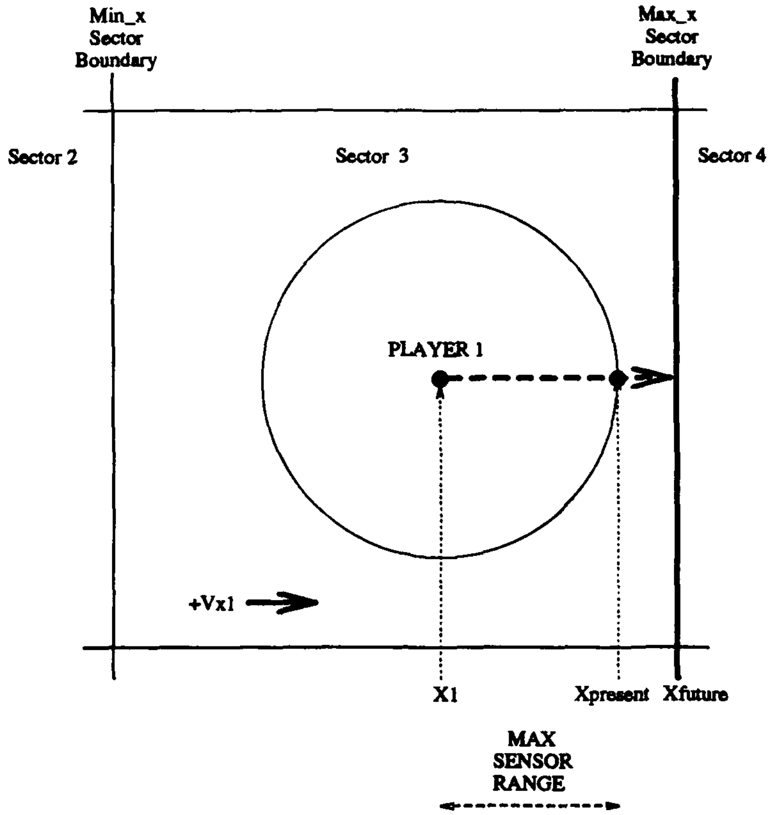

Figure 4. Valid Front End Sensor Event (Positive x-velocity)

$$t = \frac{x_{future} - x_{present}}{v_{x1}} = \frac{max\_x - (x_1 + \text{max\_sensor\_range})}{v_{x1}}$$

If realistic values of $x_1 = 5000$, $x_{present} = 8000$, and $x_{future} = 9000$ are used in this diagram, the criteria listed in situation one are met. This diagram depicts a valid FES event.

Figure 5 is a second example of a valid FES event for sector three. This time player one is traveling through sector three towards sector two. Since a negative x velocity component exists, condition two must apply for this to be a valid FES event. Player one's front end sensor, located at $x_{present}$, has not yet reached the position x_{future}. The value of $x_{present}$ for player one in figure 5 is:

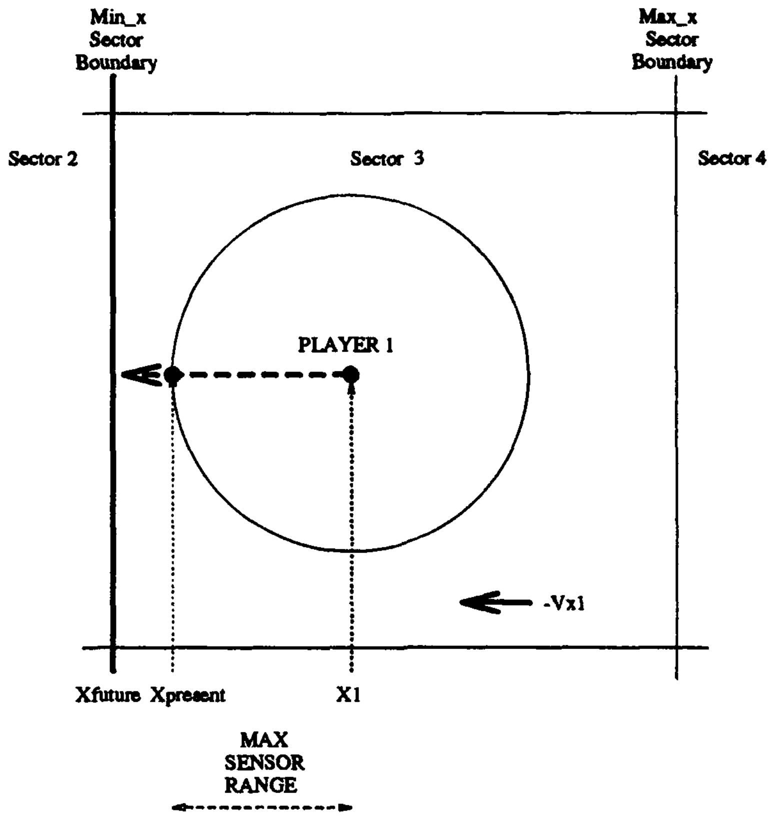

Figure 5. Valid Front End Sensor Event (Negative x-velocity)

$$x_{present} = x_1 \ - \ \text{max\_sensor\_range}$$

The location of the next sector boundary player one's front end sensor is approaching is:

$$x_{future} = \ \text{sector's minimum x coordinate}$$

Since $x_{present}$ is greater than x_{future} for player one in figure 5, this is also a valid FES event.

Figures 6 and 7 show FES events being computed for player one in sector three. In figure 6, player one is traveling from sector three to sector four with a positive x velocity component. Like figure 4, the values of $x_{present}$ and x_{future} are:

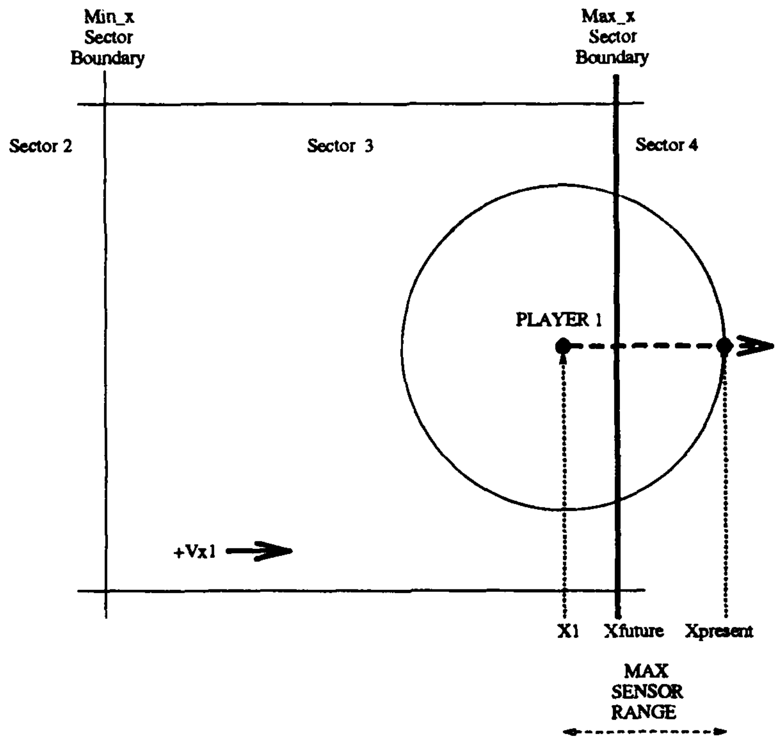

Figure 6. Invalid Front End Sensor Event (Positive x-velocity)

$x_{present} = x_1 + \text{max\_sensor\_range}$

$x_{future} =$ sector's maximum x coordinate

However, there is an important difference in this diagram from figure 4: $x_{present}$ is not less than x_{future}. This clearly indicates that *the FES event for player one in sector three has already occurred in Figure 6.* Figure 7 depicts an invalid FES event for player one in sector three as well because the value of $x_{present}$ is not greater than the value of x_{future}, with a negative x velocity component. It is important to realize that *BATTLESIM checks for a valid FES event (and BES event as well) only in the sector in which the player currently resides.* For these four diagrams, this is sector three. In order to determine the validity of a FES event for this (or any other) player in *another* sector, another check must be performed.

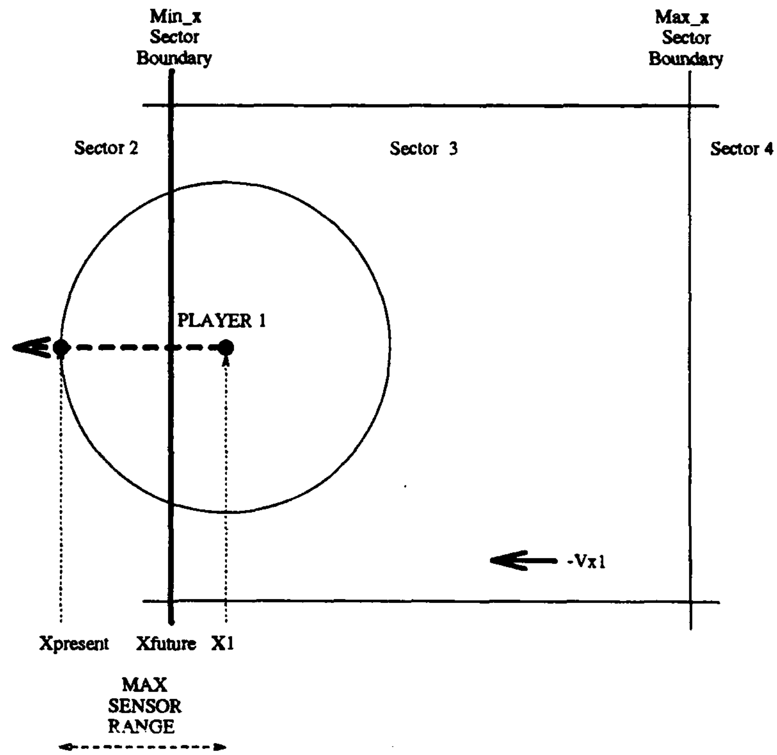

Figure 7. Invalid Front End Sensor Event (Negative x-velocity)

5.5 Center of Mass Algorithm.

The center of mass (COM) boundary crossing event is similar to the FES event; it uses the same value for the variable x_{future}. The value x_{max} is used when the x velocity component is positive because it is again the next boundary that the player's center of mass will cross, and the value x_{min} is used again when when the x velocity component is negative. However, since the object of interest is not the maximum range sensor of a player but in fact the player itself, the value of $x_{present}$ changes to the player's center of mass. This is considered to be at the player's physical center for the sake of convenience.

To compute the intersection time of a player's center of mass with the appropriate sector boundary when acceleration is zero, Equation 5 is used again. In this case, Equation 5 represents the time required for the player's center of mass to travel from its current position $x_{present}$ to the position of the sector boundary x_{future}.

While the FES event had two conditions for when it should *not* occur, a COM event has only one: it cannot have already occurred in the player's sector. The center of mass

algorithm must also be able to handle a player with a maximum sensor range of zero. This capability provides increased simulation flexibility because scenarios can then be designed with no FES and BES events.

In order to ensure a valid COM event has not already occurred in the player's sector, one of the two conditions described previously in Section 5.4 for the FES event must be true. Since the determination of whether a COM event is valid or not is similar to the FES determination already shown, examples are omitted from this section. However, the reader can refer to Appendix G for examples of valid and invalid COM events if desired.

5.6 Back End Sensor Algorithm.

The last of the three boundary crossing algorithms, the back end sensor algorithm, has been purposefully left as the last one to be discussed because it builds upon the concepts of the previous two. For instance, *a BES event can only occur for a given sector after both a FES and COM event have already occurred for the given sector.* If one or both of them have not yet occurred, then the BES event can not be the next valid event. This brings up an important point. *The BES algorithm is unique because it is checking for the player's back end sensor to exit the last sector in which the player resided, while the other two boundary crossing algorithms are checking for when their "object of interest" (either the player or sensor) leaves the current sector in which the object resides.*

As the reader may suspect, this makes the values used in the back end sensor algorithm for $x_{present}$ and x_{future} different from those used in both the front end and center of mass algorithms. The value $x_{present}$ represents the location of the back end sensor, while x_{future} indicates where the next sector boundary the back end sensor should cross is located.

The criteria for a valid BES sensor event is similar to the criteria for a valid FES event. A valid BES event is one which has not yet occurred for a player in a given sector and which is supposed to occur. A BES event should not occur when the player has a maximum sensor range of zero, or the player's BES event has already occurred in the specified sector. To ensure a valid BES has not already occurred in the specified

sector, one of the two conditions described in Section 5.4 for the FES event must be true. Since the determination of whether a BES event is valid or not is also similar to the FES determination, examples are omitted from this section as well. The reader can refer to Appendix G for examples of valid and invalid BES events if desired.

5.7 Determination of Next Boundary Crossing Event.

The previous sections have shown how to determine the validity of FES, COM and BES events in the x coordinate axis only. The reader should remember that *when determining a player's next event, BATTLESIM will always compute three potential boundary crossing events in each dimension, a total of nine events.* However, only the boundary crossing event with the minimum next event time out of the nine will be returned.

VI. Implementation

6.1 Introduction.

This chapter deals with the explanation of how all the goals and requirements for BATTLESIM were met. The first section, 6.2 is composed of three subsections describing what modifications were made to existing BATTLESIM code. Section 6.2.1 explains how the TCHSIM simulation driver was updated to the latest version, Section 6.2.2 deals with the removal of the optimistic portion of BATTLESIM, and Section 6.2.3 deals with explaining what methods and attributes had to be added to the previous player definition. Section 6.3 consists of two subsections which outline how existing data strucutures were modified. Section 6.3.1 describes what methods were added to Rizza's linked list package, and Section 6.3.2 reviews what format changes were made to scenario input files.

Section 6.4 is composed of four subsections which describe the objects added to BATTLESIM. Section 6.4.1 describes how the playerset object was generated, Section 6.4.2 describes the sector object, Section 6.4.3 shows how player messages are generated and used to transfer player state information between nodes, and Section 6.4.4 explains the array used to contain all the sector objects. Command-line arguments are covered in Section 6.5. Section 6.6 contains three subsections detailing how boundary-crossing events were implemented in BATTLESIM. They include untangling existing events (Section 6.6.1), creating boundary crossing methods (Section 6.6.2), and adding event-handling routines (Section 6.6.3). Finally, Section 6.7 describes the limitations associated with this version of BATTLESIM.

6.2 Cleanup of Existing Code.

6.2.1 Update TCHSIM Simulation Driver. As mentioned previously, TCHSIM is a simulation driver environment which provides several of the basic mechanisms required of every discrete-event simulation. This includes initialization routines, general support methods, and object-based implementations of a NEQ, event, and simulation clock (15). The implementation of TCHSIM as a discrete-event simulation environment has been ongoing like BATTLESIM.

The latest version of TCHSIM, Version 3, was not available when Soderholm accomplished his research – therefore Version 2 was used. Unfortunately, Version 2 was used in such a way that it was heavily intertwined with the BATTLESIM application code. Since the first requirement for upgrading BATTLESIM was utilizing Version 3, Version 2 had to be removed from the BATTLESIM application code.

6.2.2 Remove Optimistic Computation/Local Rollback. As previously described, Soderholm's approach relied on the use of optimistic computation of the next event on an LP in order to achieve some measure of speedup. However, the author decided not to maintain this capability because it forced the simulation to *always execute sequentially*, i.e. only one LP could execute an event at any given time. Therefore the files, flags, variables and methods necessary to implement lookahead, lookahead update, and local rollback on an LP were removed entirely. This includes:

- the files `rollback.c` and `rollback.h`
- the flags `use_lookahead` and `lookahead_complete`
- numerous global variables

6.2.3 Update the Player Object. Several changes were made to the previous player object definition. The player object is now an object-based player abstract data type (ADT). The Player ADT contains all the attributes required to completely encapsulate all the state information of a single player in BATTLESIM. Methods are provided to allow the user to retrieve, modify, and delete all attributes in a player (refer to appendix B for a complete listing and explanation of these methods). The player is defined using the `object_attributes` structure shown in Figure 8. The attributes which have been added to the structure include `sector_id`, `own_player`, `player_size`, and `player_copies` linked list. For a description of the player's attributes, refer to Appendix F.

As Chapter 4 described, a potential problem exists with handling events that have been computed locally but should execute on a different LP. There are two ways to ensure that a player gets its potential next events back from player-copies on other LPs:

```
struct object_attributes
{
        int sector_id;
        int object_type;
        int object_id;
        int object_loyalty;
        int own_player;
        double current_time;
        int fuel_status;
        int condition;
        int vulnerability;
        struct location_type location;
        struct xyz_velocities velocity;
        struct orientation_type orientation;
        struct rotation_rates rotation;
        struct operator_type operator;
        struct performance_characteristics performance;
        struct linked_list *route_data;
        struct linked_list *sensors;
        struct linked_list *armamants;
        struct linked_list *defensive_systems;
        struct linked_list *target_list;
        struct linked_list *player_copies;
};
```

Figure 8. Structure used to Define a PLAYER Object

- Send the player's potential next events back from other LPs as soon as they are computed, and place them in the destination LP's NEQ.
- Place the player's potential next events in the local NEQ where they are generated. When the event is retrieved off the remote LP's NEQ, send it back to the LP on which it really "belongs", i.e. the LP which owns the player.

The second approach is the one which BATTLESIM uses because it prevents sending messages to LPs which should not be executed. Since the conservative time synchronization paradigm employed by BATTLESIM requires that no events arrive at a destination LP with time stamps earlier than the local simulation clock, the first approach could send an event to an LP which is in the destination LP's past. This is unallowable. The second approach ensures that this situation will not occur; the conservative paradigm will advance an LP's local clock only when it know the minimum possible time stamp on messages coming from other LPs.

Sending an event from an LP which was generated by a player-copy obviously depends upon the LP knowing the event is not really its own, but "belongs" to another LP. Therefore *the LP must check to see if it owns the player which generated the event every times it retrieves an event from its NEQ.* If it does, then the LP can execute it; if it does not, then the LP must not execute the event but send it to the appropriate destination LP.

6.3 Modify Existing Data Structures.

6.3.1 Add Capabilities to the Linked List Package. Rizza designed the linked list package to be flexible and powerful enough to support any kind of data structure as elements in the linked list, and store them as First In First Out (FIFO), Last In First Out (LIFO), or priority queues (30). However, it was designed to allow a user to retrieve attributes in each linked list element only *after* the element had been retrieved from the linked list; each element is accessed like any queue, either from the front or the back. BATTLESIM requires the capability to frequently retrieve elements from the middle of a linked list to perform such tasks as copying a player from one sector to another when a center of mass event occurs (see example in Figure 9). While this could be accomplished with

existing methods by popping off all the values above the desired player until the desired player is retrieved, this technique would be extremely inefficient and processor intensive. Instead, the linked list package needs the ability to retrieve an element *anywhere in the linked list.* Also, the ability to do a low-level copy of each linked list without regards to what kind of structure is being copied would allow the application to be more object-based, with the higher level data structures having no knowledge of the format of the lower-level data structures. The methods added to BATTLESIM to support these new capabilities include:

- **ll_get_data** - Returns the top item (which contains both a data entry handle and a handle to the next linked list element) from a specified linked list, but does not delete it from the linked list like ll_pop.
- **ll_get_ptr** - Returns the handle to the top of the provided linked list element. This is used to begin traversal of the linked list looking for a particular element.
- **ll_next_data** - Returns the next element in the linked list *after* the one provided, without deleting it.
- **ll_next_ptr** - Returns the handle to the next linked list element *after* the one provided. This method is called after calling ll_get_ptr in order to continue traversal of the linked list looking for a particular element.
- **ll_copy** - Copy the contents of any linked list to another linked list *verbatim* without regards to whether the source was a LIFO, FIFO, or priority queue.
- **change_typell** - Permits the user to change the type field of a specified linked list.

6.3.2 Change Format of BATTLESIM Scenario Input Files. The format for the single scenario file used by the previous version of BATTLESIM was too limited. No information was labeled and comments were not allowed, making it extremely difficult to find out what values particular attributes were being initialized to — not to mention that all necessary attributes had been entered and in the proper sequence.

Several revisions have been made to the format of the scenario files used by BATTLESIM. First of all, a new file was created to contain all code related to reading in the

scenario files. Each scenario file has a version number associated with it, which is used to ensure that the information in the scenario file matches the order and number of attributes expected. Comments may be placed anywhere in the scenario file, either on their own lines or on lines containing information, by starting them with an asterisk; everything in a line past an asterisk is considered a comment. Route points can be entered in the order they should be reached, instead of backwards like before. The version number is contained in the file name extension, like 'bench130.in4'. In this case, the extension 'in4' indicates that this scenario file is version 4, the current version expected and required to be a valid scenario file in BATTLESIM.

The format for scenario files with the '.in4' extension is listed in Appendix F. Each scenario file contains two general kinds of attributes: those which are common to all players defined in the scenario file, and those which are unique to each player. For instance, the attributes 'version number' through 'icon definitions' are common to all players defined in the scenario file. Thus they are defined only once. In contrast, the attributes 'object type' through 'defensive systems' are unique to each player whose initial state information is provided in the scenario file. Therefore, these attributes are repeated once for each player being defined.

It is vital that the scenario generator ensure that each player is defined only in the scenario file which initializes the LP responsible for that player's initial route point. For example, if player 3's initial route point is located in sector eight, and sector eight is controlled by LP 2, then LP 2's scenario file should be the only file containing the initial state information for player 3. If this is not done, then abnormal BATTLESIM execution may occur.

6.4 Create New Objects.

6.4.1 Create the Playerset Object. The playerset object is an object-based ADT which contains all the players visible in a given sector. The players located in a sector can be there for one of two reasons:

- The sector "owns" the player. This means that the player physically resides within this sector.
- The sector does not "own" the player, but instead possesses a player-copy. In other words, *every player is owned by exactly one sector in the simulation. This is done to allow a player, even though it physically resides in one sector, to perform sensor checks in other sectors that it has visibility.*

The playerset contains no attributes which are specific to the playerset — its *only* purpose is to serve as a "container object" to hold all the players that are visible in a sector, i.e. they are owned by that sector or just have visibility there. Each sector contains one playerset. Playerset methods are provided to allow the sector to retrieve, modify, and delete all the players in a playerset (refer to appendix B for a complete listing and explanation of these methods).

As mentioned earlier, Soderholm used an array called the "master object array" to keep track of all the players in the current scenario. This structure was not maintained in the current version of BATTLESIM for several reasons:

- Since it was statically allocated to hold 1024 entries, each of which pointed to a player structure, a given scenario could only contain a maximum of 1024 players.
- The array was not designed to support a partitioned battlefield, i.e. it did not have the ability to determine which players belonged to which sectors since Soderholm's version of BATTLESIM did not incorporate sectors.
- Searching for a particular player using Soderholm's algorithm required $O(n)$ time, not a particularly efficient search time.

Since a BATTLESIM sector requires frequent access to its playerset in order to access and update its players, a more advanced data structure with a lower search and update access time was deemed appropriate. The playerset structure designed for implementation was an array with 25 entries, with each entry acting as a "bucket" which holds a set of entries unique to that bucket. This structure, commonly known as an ***open hash table***,

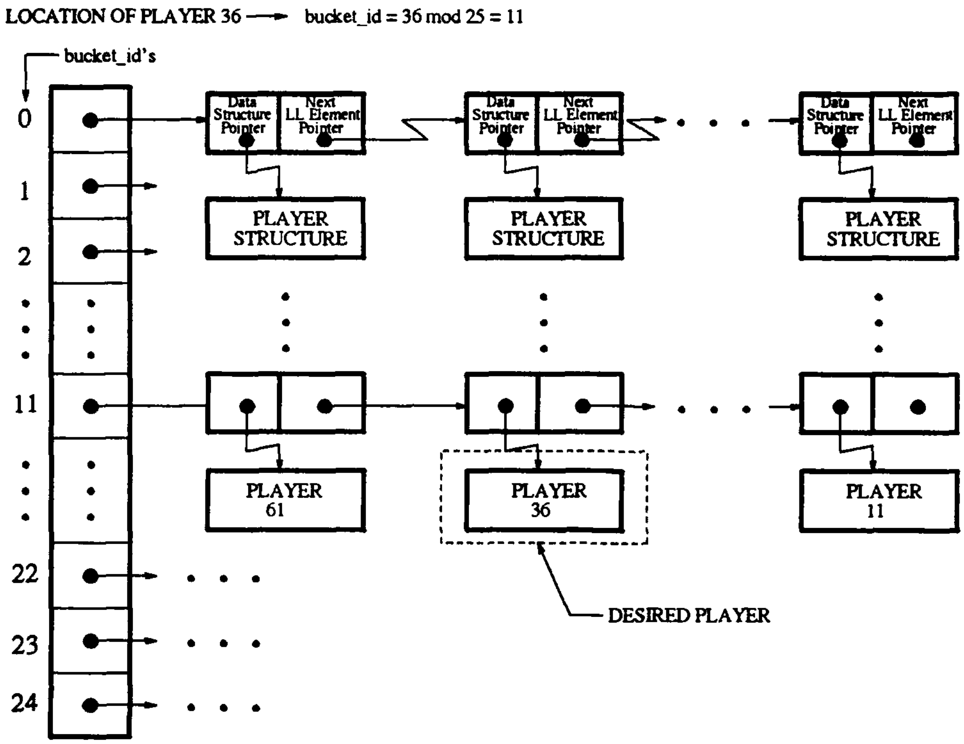

Figure 9. Depiction of a Playerset as an Open Hash Table

distinguishes each bucket with identifiers sequentially ordered from 0 to 24; each bucket is implemented as a linked list. Each bucket in the hash table contains a handle implemented as a void pointer to its respective linked list, with that linked list containing all the players which meet the hashing function criterion for that bucket. In the case of BATTLESIM, the hashing function used to determine which players go in which bucket is:

$$\texttt{bucket\_id} = \texttt{player\_id mod 25} \tag{6}$$

An example depicting a typical playerset access is depicted in figure 9. In this example, BATTLESIM is locating a player structure with an `object_id` value of 36. The first step in updating this entry is to locate which bucket it belongs by using the hashing function described in equation 6.

$$\texttt{bucket\_id} = \texttt{player's object\_id mod 25} = \texttt{36 mod 25} = \texttt{11}$$

This indicates the player is located in bucket 11 of the playerset's hash table. The linked list belonging to this bucket is traversed until the player structure containing the `object_id` value 36 is found. The handle to this player structure is returned to the sector, which then updates the player structure directly using predefined player methods.

There are several advantages to using a hash table instead of the master object array for tracking players in a BATTLESIM scenario.

- The number of players allowed in a scenario is no longer limited to 1024, but is only limited to the number that can be supported by the amount of memory on each node in the Hypercube.
- Memory is no longer statically allocated to hold the scenario's players. Instead, memory is dynamically allocated as it is needed to add more players and dynamically deallocated as players are removed.
- The use of a separate playerset for each sector naturally supports the ability to distinguish which players belong to which sectors by merely checking the contents of a sector's playerset.
- On the average, the number of entries that must be searched to find a particular player is $\frac{n}{b}$ instead of n. Since selecting the correct bucket using the hashing function described earlier requires $O(1)$ time, searching in the playerset's open hash table only takes on the order of $O(1 + \frac{n}{b})$ time (2:220). On the average, this will result in access times which will improve as the number of players in the playerset increases. Note that access time for a given player approaches $O(1)$ as the number of buckets in the hash table approaches the number of players in the sector.

6.4.2 Create the Sector Object. The sector object is an object-based ADT which contains all the attributes required to completely encapsulate a single sector in BATTLESIM. Methods are provided to allow the user to retrieve, modify, and delete all attributes within a particular sector (refer to appendix B for a complete listing and explanation of these methods). The sector object definition is defined by the structure **`sector_struct`** shown in Figure 10.

```
struct sector_struct
{
        int sector_ID;
        double min_x_value;
        double max_x_value;
        double min_y_value;
        double max_y_value;
        double min_z_value;
        double max_z_value;
        void *Pset;
        struct neighbors neighbor;
        double next_event_time;
};
```

Figure 10. Structure used to Define a SECTOR Object

The purpose of each of the attributes in Figure 10 are as follows:

- sector_ID - an integer which uniquely identifies this sector in the battlefield
- min_x_value - the minimum x coordinate of the sector
- max_x_value - the maximum x coordinate of the sector
- min_y_value - the minimum y coordinate of the sector
- max_y_value - the maximum y coordinate of the sector
- min_z_value - the minimum z coordinate of the sector
- max_z_value - the maximum z coordinate of the sector
- *Pset - a handle to the sector's playerset. It is a void pointer since the sector does not need to know what structure the playerset has — it only needs the playerset methods to access and modify it.
- neighbor structure - a structure containing the sector_ID's of all the sector's which physically border this sector (additional information on this feature follows).

- next_event_time - attribute indicating the minimum next_event_time of all the players in the sector. Added to support future BATTLESIM upgrade.

One of the problems that must be addressed when partitioning the battlefield is the possibility of a player moving outside of the battlefield's predefined boundaries. This might occur when a player evades another player in the battlefield because the other player is an enemy which has not yet sensed him. Three possibles courses of action exist:

1. Stop the simulation, returning a message indicating the simulation terminated, and why.
2. Continue the simulation, but move the player to a location in the same sector which no longer causes it to move outside battlefield boundaries.
3. Continue the simulation by acting as if the simulation's sectors were connected to one another in a "wrap-around" fashion. This would allow the player to fly out of the current sector and into another one.

Option one was dismissed since the last thing desired is to stop the simulation when a valid course of action (like evade) occurs. The second option was likewise dismissed since player movement would no longer be modeled smoothly; the player would have to be "picked up" and laid back down in another part of the sector. Option three offers the advantage of allowing the x and y dimensions of the battlefield to be conceptualized as a sphere, with all sectors connected to other sectors in a wraparound fashion (see Figure 11). This allows a player to continue moving unaffected in all directions as if there were an infinite number of sectors in the battlefield, with no knowledge of whether it is exiting the physical dimensions of the battlefield or not. This approach is realistic for modeling since a player does not "fall off" the edge of the world when traveling either.

In Figure 11, a battlefield consisting of 20 sectors is shown with lines depicting where sectors wrap-around. Players can move diagonally as well; however, the arrows depicting diagonal movement have been intentionally omitted to keep the diagram straightforward.

In order to use wraparound during a given run, the simulation has to know which sector the player is going to enter. One of two techniques can be employed:

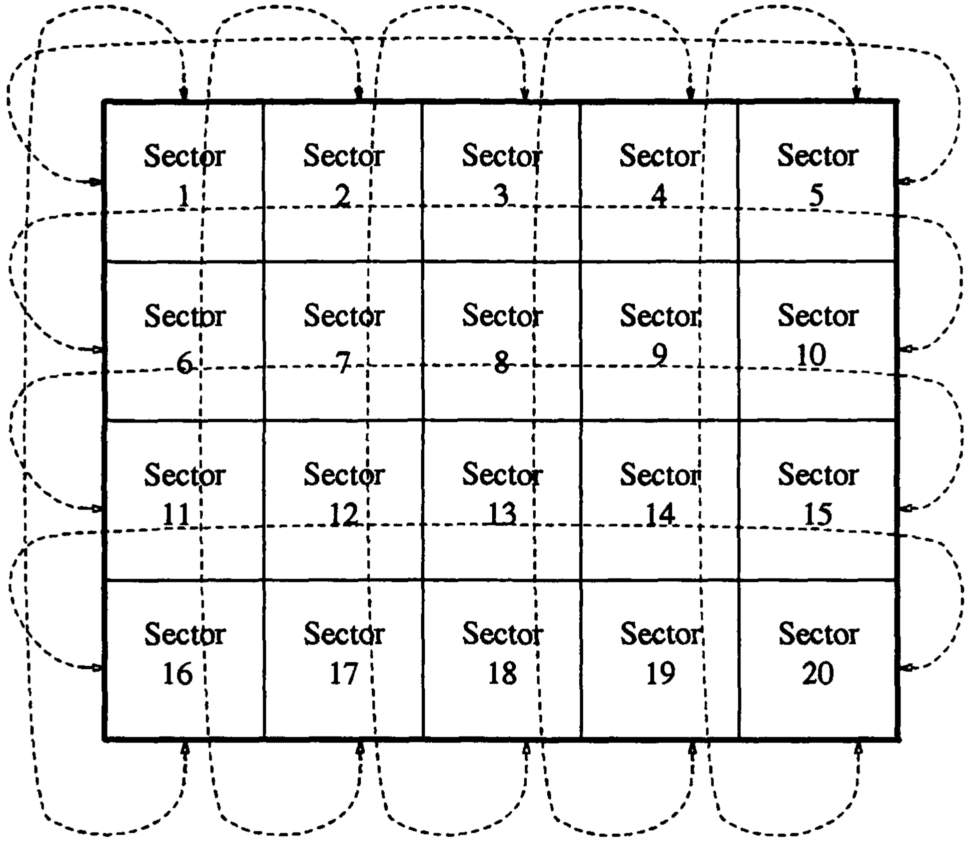

Figure 11. Player "Wrap-around" on Battlefield

- Recompute the neighbors of the specified sector each time it is needed when wrap-around occurs.
- Compute the neighbors of each battlefield sector once at the beginning, and then store that information for potential use later.

Sector locations are provided by scenario files and remain fixed for the life of the simulation run, making option one compute the same values over and over again. Since memory utilization is not a problem, option two was chosen. The *neighbors structure* in the sector ADT tracks all the neighbors of a given sector. Since each sector is defined as a rectangular cube, each sector is potentially bounded by up to 26 neighbor sectors. The naming convention for these attributes uses the letter 'o' to signify 'over', (the sector is on level 1), 'u' to signify 'under', (the sector is one level 3), and neither of these if the sector is on the same level as the provided sector (level 2).

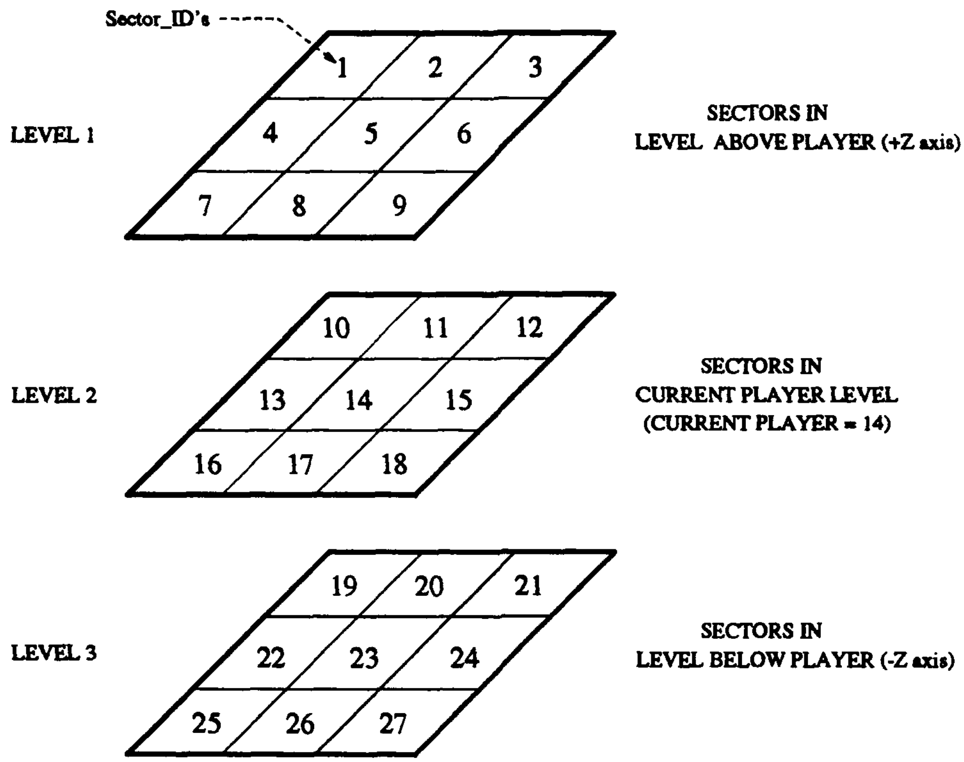

Figure 12. Determining the "Neighbors" of Sector 14

In order to better grasp the spatial relationship of sector neighbors to attribute definitions, figure 12 illustrates where each of these sectors are located respective to sector 14, our sector of interest in this example. Sector 14's neighbors, which are identified by unique integer values representing sector IDs, are shown in table 1:

The present version of BATTLESIM only uses sectors on level 2 in figure 12 since sectors extend from the bottom to the top of the battlefield in the z axis. Therefore dummy values of -1 are currently generated for those sector entries on levels 1 and 3. The attribute definitions for sectors on levels 1 and 3 were added to the neighbors structure now as "hooks" for future BATTLESIM development.

6.4.3 Create Player Message Object. As previously mentioned, the Intel iPSC/2 Hypercube is a *distributed memory* parallel computer, i.e. each node has its own memory which no other node in the Hypercube can access directly, thereby forcing the use of some mechanism to pass information between nodes. The Hypercube utilizes the concept of

Table 1. The Neighbors of Sector 14

Neighbor	Sector Nbr	Neighbor	Sector Nbr
Left	Sector 13	**Right**	Sector 15
Top	Sector 11	**Bottom**	Sector 17
Top_left	Sector 10	**Top_right**	Sector 12
Bottom_left	Sector 16	**Bottom_right**	Sector 18
Directly_over	Sector 5	**O_left**	Sector 4
O_right	Sector 6	**O_top**	Sector 2
O_bottom	Sector 8	**O_top_left**	Sector 1
O_top_right	Sector 3	**O_bottom_left**	Sector 7
O_bottom_right	Sector 9	**Directly_under**	Sector 23
U_left	Sector 22	**U_right**	Sector 24
U_top	Sector 20	**U_bottom**	Sector 26
U_top_left	Sector 19	**U_top_right**	Sector 21
U_bottom_left	Sector 25	**U_bottom_right**	Sector 27

message-passing in which messages containing information are sent from a source node and subsequently received at a corresponding destination node. *Message-passing on the Hypercube can only take place if the message to be transmitted first exists in a contiguous block of memory on the source node.*

In the previous version of BATTLESIM, Soderholm sent two types of messages between nodes:

- Event Messages - contained an entire event definition in the format specified by TCHSIM (15). These were passed between nodes using SPECTRUM. Since it was designed to only handle one message type, in this case an event message, SPECTRUM was not used to transfer any other types of messages.
- Player Messages - contained the entire contents of a player structure. These messages were created using three methods especially developed by Soderholm to pass player messages from one node to another without the use of SPECTRUM.

The current version of BATTLESIM also needs the ability to send player messages between nodes, e.g. when a player is being moved from one sector to another with each sector controlled by different LPs. While Soderholm's method worked in the case of a player structure, it was not designed to be as generic as possible.

The current version of BATTLESIM generates a player message in a generic manner which can be applied to any kind of data structure. The methods created to build player messages and receive them are contained in the file **message.c**. Some of the objectives in creating the new player message transmission package included:

- If dynamically sized structures (like the player's linked lists) exist in the information, then determine the total number of bytes in the information to be transmitted. Send that value in an initial message to the destination node so it can allocate space for the information before it is received. Then send a second message containing the information itself.
- Transmit all information, whenever possible, as a single message versus several smaller messages. Since every message requires extra attributes (like size, destination node, message type), reducing the number of messages sent reduces the number of bytes required, and should reduce total message-passing time.
- Pack all information into a single contiguous block of ASCII characters devoid of any structure and field references. This means that values referenced by handles must be retrieved and placed in the message directly, since handles are useless in different address spaces.

Since structures and fields references are omitted from the message, both the source and destination nodes must have knowledge of the structure being transmitted. Otherwise the receiving node cannot determine the structure of the information passed in the message.

The methods created to support transmission of player messages include:

- **packPmessage** - Builds a message in contiguous memory containing the entire contents of a specified player, as well as extra information necessary to unpack the message on the receiving node. The method consists of three subordinate methods:
  1. pack_ll_sizes - Packs an integer for EACH of the player's linked lists into the beginning of the player message. The integer represents the number of entries in a particular linked list.

2. pack_Pmsg_player - Packs all the attributes inside a player (except for linked lists) into the middle of the player message.

3. pack_linked_lists - Packs each of the specified player's linked lists into the end of the player message.

- **listPmessage** - Lists all the attributes contained within the player message, in order to verify the message was generated properly. The message is left unmodified.
- **unpackPmessage** - Unpacks a player message containing the entire contents of a player, and either updates or creates a player on the destination node based upon the attributes unpacked. When this method is finished, the player on the destination node will match the player in the message, with the exception of the ownership attribute.

The format of the player message is shown in Figure 13. In this diagram, six integers are the first entries in player message. These entries represent the number of entries in each of the player's linked lists. Next, the actual contents of the player structure itself resides in the player message in two main parts — that part of the player excluding the linked lists, and the linked lists themselves. The first part consists of a total of 21 integer and 13 double attributes, even though only 16 "attributes" are depicted in figure 13. This is because some of those "attributes" are in fact data structures; they exist in the player message as their corresponding attributes. This is allowable since the **unpackPmessage** method knows the order and type of each attribute in the player message. Unlike all the attributes that precede them, the space required to hold the player's linked lists is dynamic — it depends upon how many entries each linked list contains.

6.4.4 Create Sector Container Object. Now that ADTs are defined for players, playersets, sectors and player messages, some kind of "container object" is required to hold all the sectors which together comprise the entire battlefield. The three structures considered for implementation are an array, a linked list, and a hash table.

Since the number of sectors in a given scenario is not known until execution time, a structure containing just enough space to hold all those sectors cannot be determined

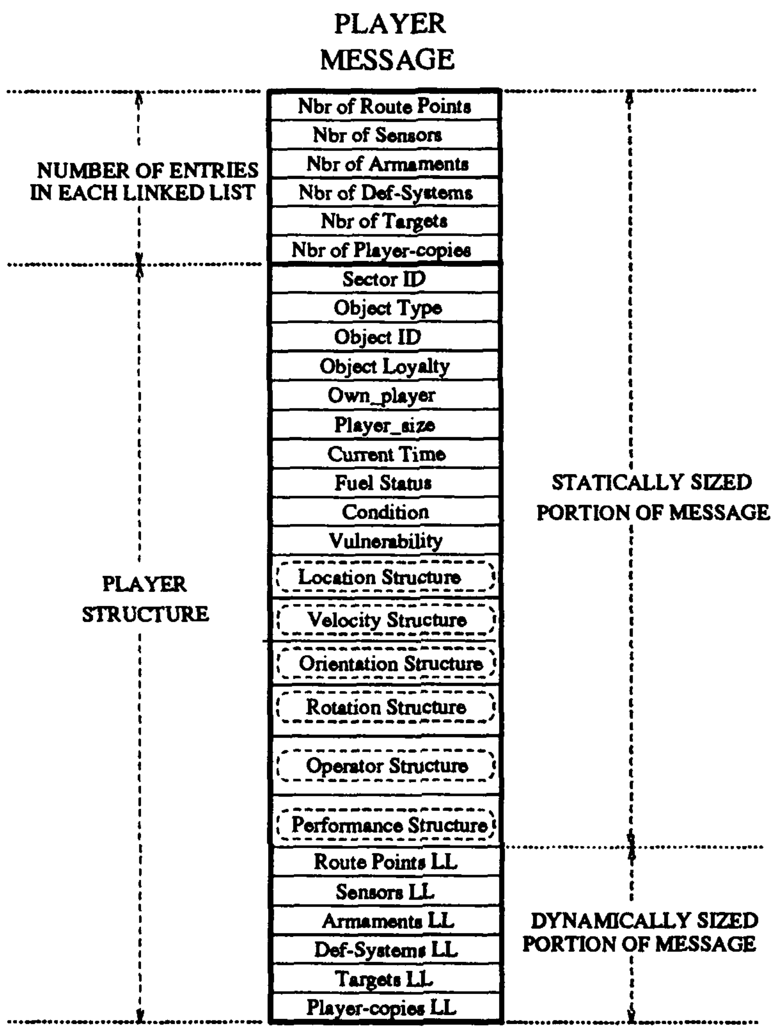

Figure 13. Structure of a Player Message

a'priori. This would tend to encourage application of a linked list or hash table. However, the author decided to use an array for the following reasons:

- The range of the number of sectors desirable in scenarios would tend to be less than the number of players. For the purposes of this research, 64 is a practical upper limit.
- The amount of storage necessary to hold all the information in a sector structure is static. This was not true for the player structure, since it contained 6 linked lists.

Therefore a sector array, initialized to hold 64 sectors and called *sector*, was added. The sector array was *not* implemented as a separate ADT because the author believed

that this added an additional abstraction layer which did little to enhance the application's future maintainability. Each sector is indexed in the array by its sector_ID: sector 1 is in array entry sector[1], sector 2 is in array entry sector[2], etc. to make it easy to remember where each sector is kept. A variable called SECTOR_ARRAY_SIZE exists to change the number the number of sectors quickly if more sectors are desired later.

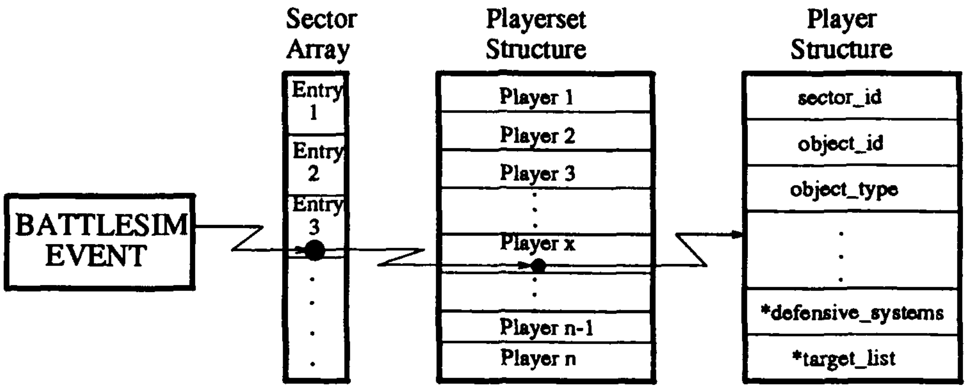

Figure 14. How BATTLESIM Retrieves a Player

Figure 14 shows how attributes in the player, playerset, and sector objects are used to find a particular player in a specific sector. Each BATTLESIM event contains several attributes including the player's object_ID and current sector. Since an event knows which sector the desired player is located in, it goes to the proper sector array entry. The sector array entry contains a handle to the sector's playerset. The playerset contains a handle to the desired player, which is found using the player's object_ID (see Figure 9). The player can now be retrieved using its handle.

6.5 Add Command-Line Arguments.

One of the design goals outlined in Chapter 3 was the ability to change configurations easily. The addition of several new features to the present version of BATTLESIM requires an effective method for providing information to the simulation without extensive scenario regeneration and recompilation of code. Command-line arguments provide this capability.

Before showing what options are available and how they are used, some explanation of naming conventions in BATTLESIM is necessary. *The current version of BATTLESIM*

is designed to accept files only with the .in4 extension; previous formats are incompatible and cause erroneous execution. This extension name, in which the integer value increases as changes are made to the scenario file format, was designed to identify the scenario file's format. If multiple input files are being used for a simulation run, then the files must be created with the following naming convention:

scenario file name format = < **scenario_name** >< **LP_identifier** > **.in4**

where

scenario_name = name unique to this set of scenario files

LP_identifier = integer identifer of LP which uses this file

For example, benchmark 13, described in detail in Appendix A, is designed to run on 8 LPs. The names of its 8 files are:

bench130.in4 - specifies bench13, LP0 scenario file with .in4 format

bench131.in4 - specifies bench13, LP1 scenario file with .in4 format

.

.

.

bench136.in4 - specifies bench13, LP6 scenario file with .in4 format

bench137.in4 - specifies bench13, LP7 scenario file with .in4 format

All benchmark scenarios for BATTLESIM (benchmark scenario 13 is described in Appendix A) have naming conventions similar to this. The only differences are the benchmark number and how many scenario files exist, e.g. if the scenario was designed for just 4 nodes, then it would have only four files with LP_ids of 0 to 3. Lastly, all files specifying sector-to-LP assignments must end with a **.map** extension in order to be valid; BATTLESIM will reject them otherwise. The format of a command-line argument in BATTLESIM is:

$$\text{command} - \text{line argument format} = - < \textbf{switchidentifier} >< \textbf{argument} >$$

where

switch identifier = a single letter which indicates the argument's purpose

argument = the data portion of the command-line argument

All of the command line options to be used in a given scenario run must be entered together. BATTLESIM supports seven options total. These include:

- **-S or -s switch** - specifies the use of a *single* input scenario file. To do so, the user immediately follows the switch with the scenario filename (excluding the .in4 suffix on the filename). This file will be read by all LP's in the scenario to initialize their sector array state information.
- **-M or -m switch** - specifies the use of *multiple* input scenario files. To do so, the user immediately follows the switch with the scenario filename (excluding the LP identifer and the .in4 extension). Each LP will read only its own scenario file.
- **-P or -p switch** - specifies which sector-to-LP mapping file is to be used. The required extension .map must not be entered.
- **-D switch** - specifies variables that are to be defined at run time. This was added to allow print statements controlled by debugging variables to be activated and deactivated at run time. If there is more than one variable to be defined, each one must use its own -D switch.

Since each run requires that BATTLESIM have access to a scenario and map file, defaults for these two entries exist. For instance, BATTLESIM uses the single scenario file **`datafile.in4`** if the user does not specify any scenario files. It also defaults to the sector-to-LP mapping file of **`battlesim.map`** if no other is provided.

6.6 Implement Boundary Crossing Events.

6.6.1 Untangle Existing Events. The previous version of BATTLESIM contained a total of six event types which were described in Chapter 4:

- Reached_Turnpoint
- Entered_Sensor_Range
- Made_Sensor_Contact
- Ordnance_Released
- Ordnance_Reached_Target
- Collision_Distance_Reached

Once the initial Reached_Turnpoint events had been scheduled by BATTLESIM, determination of all subsequent events (except for Ordnance_Released) was handled by performing a sensor check. This resulted in an unusually large and complex method for sensor check — one which made it difficult to understand and update with the three new boundary-crossing events. An easier layout was in order.

All of the steps previously handled by Sensor_Check — except Ordnance_Released and Ordnance_Reached_Target — are now encapsulated with their own supporting methods. To maintain compatibility with existing BATTLESIM code, the Sensor_Check call was left intact, but it now makes calls to new methods segregated by function. The new methods, including those supporting the three new boundary-crossing events, are:

- **Det_Next_Event** - determines what the next event for a given player is, and adds it to the appropriate LP's next event queue. It calls the following subordinate methods:

    1. Det_Route_Event
    2. Det_Boundary_Event
    3. Det_Collision_Event
    4. Det_Sensor_Contact

5. Det_Other_Contact

Each time Det_Next_Event calls a subordinate method to determine the next event of a particular type, it keeps track of the event with the minimum time of all those checked up to that point. When all subordinate methods have been called, Det_Next_Event knows which event type has the minimum next event time; it moves the event associated with that time to the LP's next event queue.

- **Det_Route_Event** - Computes the next potential Reached_Turnpoint event for a player, and returns it as a temporary event.
- **Det_Boundary_Event** - Computes the next potential boundary-crossing event for a player, and returns it as a temporary event. The event returned is either a FES, COM, or BES event.
- **Det_Collision_Event** - Computes the next potential Collision_Distance_Reached event for a player, and returns it as a temporary event.
- **Det_Sensor_Contact** - Computes the next potential Made_Sensor_Contact event for a player, and returns it as a temporary event.
- **Det_Other_Contact** - Computes the next potential Entered_Sensor_Range event for a player, and returns it as a temporary event.

6.6.2 Create Boundary-Crossing Methods. The method Det_Boundary_Event was further subdivided into determining the next boundary-crossing event in each of the x, y, and z coordinate axes due to the size and complexity of each one (see Chapter 5 for an in-depth discussion of the boundary-crossing algorithms implemented). The methods associated with determining these boundary-crossing events include:

- **Time_to_Intercept_Bound** - Computes the minimum time for a player to intercept either the x, y, or z-axis boundary of a given sector, and returns the event associated with it as a temporary event. It calls the next three methods.
- **Time_to_Intercept_X_Bound** - Computes the minimum time for a player to intercept the x-axis boundary of a given sector, and returns it as a temporary event.

- **Time_to_Intercept_Y_Bound** - Computes the minimum time for a player to intercept the y-axis boundary of a given sector, and returns it as a temporary event.
- **Time_to_Intercept_Z_Bound** - Computes the minimum time for a player to intercept the z-axis boundary of a given sector, and returns it as a temporary event.

Since only one of these events is added to an LP's next event queue every time the method Sensor_Check is called, five out of the six potential next events will go unused. Therefore it is likely that several of these events will still apply and be returned the next time Sensor_Check is called.

6.6.3 Add Boundary-Crossing Event Handling Routines. The methods described in the previous section provide the capability to add the appropriate next event for a given player to the player's next event queue. However they do not control *what is supposed to happen* when these events are removed from the next event queue to be executed. That is the purpose of the event handling routines. This section describes the three boundary-crossing event handling routines added to BATTLESIM — front end sensor (FES), center of mass (COM), and back end sensor (BES) — by describing how they work and what calls they make to other supporting methods.

The purpose of the FES routine, as previously mentioned, is to replicate all state information associated with a player — through the creation of a player-copy — from one sector to another because the player's maximum range front end sensor has crossed the boundary separating the two sectors. This information is needed in the sector "gaining" the player-copy because:

- The gaining sector uses the player-copy to determine whether the player has any sensor events associated with other players in the gaining sector.
- It allows the simulation to remove the player later from the "losing sector" when ownership is tranferred from the losing to the gaining sector.

While the player is copied in its entirety, the 'own_player' attribute of the new player-copy is automatically set to false since the player-copy added in the gaining sector is just that

```
Extract Event Structure/Player Info

Add gaining sector ID to Player-copies linked list

player_message = packPmessage(player)

IF (gaining_sector on same node as original sector)
   copied_player = unpackPmessage(player_message);
   add_Pset_player(gaining_Pset, copied_player);
ELSE
   destination_node = node containing gaining sector
   Send message containing player_message size
   Send player_message

Update all player-copies with proper time, position, and velocity

Determine Next Player Event
```

Figure 15. Front End Sensor Event-handling Routine Pseudocode

— *a player-copy that is not owned by the gaining sector.* It is noteworthy to remember that if the maximum sensor range of the player in question is zero (meaning the player has no sensing capability whatsoever), then this method will never be called. The FES is not even considered because a player is only copied from sector to another *when it has the ability to sense or collide with an object in a sector other than the one in which it currently resides.* In the case of a zero sensor range, this is only true once the player's center of mass crosses a sector boundary. The pseudocode describing the main steps followed to replicate the provided player from its current sector to the gaining sector in Figure 15.

Since a new copy of the player is being added to the gaining sector's playerset, the player needs to update its player-copies to ensure that it maintains accurate information on where all of them are located. The three boundary-crossing events all use the following sequence of steps to update a player's player-copies:

- First update the player since all events act upon and are determined by it.

- Then update all other player-copies of the player, the ones not owned, by **replacing** them completely with a mirror image of the owned player-copy.
- Then the player is packed into a player message suitable for sending to a node on a network.

If the gaining sector resides on the same Hypercube node as the original sector does, then the player-copy can be added to the gaining sector directly in memory since all resources on the same node share a common memory. The player-copy is unpacked from the message and added to the gaining sector's playerset directly using the playerset's handle. But if the gaining sector resides on a different node, then the memory containing each sector's state information is not shared, and the player message must be sent. BATTLESIM calls TCHMAP to determine which LP — and thus which node — this sector belongs to. It then generates two messages for transmission to the other node: one to indicate how big the second player message (containing the player copy) is, and the player message itself. The first message is required since the second message can vary widely in size due to the player's linked list structures; the receiving node has to allocate space for the new player copy it is receiving *before* it arrives. Unforunately, since extensions to SPECTRUM have not been completed to support passing BATTLESIM messages between nodes, print statements are used to represent message transmission at this time.

The correct execution of BATTLESIM relies on each player-copy in the simulation having access to completely reliable state information. Otherwise, invalid computations such as sensor events may be generated. The last two steps in the pseudocode ensure that all of the copies *which already existed* of the player just copied — regardless of where they are at — are kept up to date. The method Update_Position_2 updates all the owned player's attributes which should change value as a result of the COM event (player position, velocity, and orientation), and then calls Send_Pcopy_Updates to send the update player to wherever player-copies already exist and replace them. The method Sensor_Check must be called last in order to determine what is the next valid event (if any) for this player and add it to the appropriate LP's NEQ.

```
Extract Event Structure/Player Info

max_range = player's maximum sensor range

IF (max_range = 0) AND (player-copy not in gaining sector)
   Perform Steps of Front_End_Sensor

IF (player-copy in gaining sector)
   own_player(gaining sector player-copy) = TRUE
   own_player(losing sector player-copy) = FALSE
   Add gaining sector ID to player's player-copies linked list
   Update event to point to new player-copy in gaining sector
ELSE
   Output an error message saying player not already in gaining sector

IF (max_range = 0)
   Perform steps of Back_End_Sensor

Update all player-copies with proper time, position, and velocity

Determine Next Player Event
```

Figure 16. Center of Mass Event-handling Routine Pseudocode

The purpose of the COM event-handling routine, as specified earlier, is merely to transfer ownership of a given player identifier from one sector to another because the player's center of mass has crossed the boundary separating the two sectors. The routine itself is rather interesting because it not only has to handle its own unique tasks, but also those associated with the FES and BES events when the player associated with the event has a maximum sensor range of zero. This is due to the requirement that BATTLESIM be able to sustain execution with players possessing such a characteristic (see Chapter 3). The pseudocode describing the main steps in COM are depicted in Figure 16.

The COM routine, like the FES routine which sometimes precedes it, is passed an event from which the routine extracts information. Some of the primary information retrieved includes:

- The object_id of the player whose ownership status is changing.
- The sector identifier of the sector losing ownership of the player.
- The sector identifier of the sector gaining ownership of the player.
- The range of the player's farthest reaching sensor.

The next major task to accomplish depends upon whether the FES routine has already occurred for this player while the player-copy in question travels from the sector losing ownership to the sector gaining ownership. If the FES event-handling routine has occurred, then the player-copy already exists in the gaining sector; the steps in the FES routine to copy the player do not need to be accomplished again. If not, then the player-copy may not exist in the gaining sector yet, so a check is made. If the player is indeed not there from an earlier FES or COM routine, then the steps that *would* have been accomplished by a FES routine — except for Update_Position_2 and Sensor_Check — are duplicated in the COM routine.

The routine knows that the gaining sector does not yet own the player, so it double-checks to ensure the player-copy exists in the gaining sector like it should by now. If it doesn't, an error message is output; if it does, then transfer of player ownership is transferred from the player in the losing sector to the player in the gaining sector. It is important to realize that *the player is not being removed yet from the "losing" sector because it still has visibilility in the sector it just departed.* The player in the gaining sector then updates its player-copies to reflect that the player now exists in that sector.

Once this has been accomplished, the attributes in the event passed to the COM routine are updated to reflect the new location of the player since *the next event for a player is always determined from the "owned copy" of a player.* The COM routine then checks to see if the player's maximum sensor range is zero again, because it needs to know whether the player can actually sense back into the previous boundary. If the value is zero, then the player is removed entirely from the sector which just lost ownership because the plane can no longer sense anything in it — the only reason for a player-copy to be maintained there. The steps in the BES routine — except for the calls to Update_Position_2

```
Extract Event Structure/Player Info

IF (player in losing sector)
   Remove player from sector
ELSE
   Print error message

IF ( length(player's player-copies linked list) < 2 )
   Print error message
ELSE
   Remove losing sector from player's player-copies linked list

Update all player-copies with proper time, position, and velocity

Determine Next Player Event
```

Figure 17. Back End Sensor Event-handling Routine Pseudocode

and Sensor_Check — are executed. Otherwise, the steps associated with the BES routine are skipped.

The final steps for the BES routine are like those of the FES routine. The player's player-copies are updated to match the player precisely. Lastly, the next event for the player is added to the next event queue owned by the LP controlling the player.

The last boundary-crossing routine to be discussed is the BES event-handling routine. Its purpose, as previously described, is to remove an existing unowned copy of a player from a sector completely because the player no longer has visibility into that sector. Like the FES routine, this routine will *never* be executed for a player if that player's maximum sensor range is zero; its steps will be handled by the COM routine instead.

The pseudocode describing the main steps followed to remove the player-copy from the losing sector is depicted in Figure 17.

The Front_End_Sensor routine is passed an event from which the routine extracts information like all the other events. This information includes:

- The object_id of the player being removed from the losing sector.
- The losing sector identifier.
- The current sector identifier.

The attribute used to hold the losing sector identifier is different from the one used to holding the gaining sector identifier for the FES event. If the same attribute had been used, then the COM routine would not have been able to perform the steps of both the FES and BES routines because it wouldn't have both the losing and gaining sector identifiers required to mimic them.

Once the information has been retrieved, BATTLESIM checks to make sure the player exists in the losing sector like it should. If not, it outputs an error message; otherwise, it removes the player-copy from the losing sector. Next, the player's status must be updated so that it is accurate before all of its player-copies are updated. This is accomplished by removing the player-copy from the player's player-copy list, calling Update_Position_2 to update the rest of its attributes, and sending those updates to the remaining player-copies. Finally, the Sensor_Check routine is called to determine and add the next event for the player to the appropriate NEQ.

6.7 Implementation Limitations.

While the current version of BATTLESIM provides significantly more flexibility than the previous version did, limitations in how it is used must be enforced or proper execution may not occur.

A scenario may not be generated in which any player has a sensor range which extends into more than just one sector past the player's current sector. The limitation resides in SPECTRUM, the mechanism used to pass messages between nodes. Since SPECTRUM relies upon the use of 'arcs' of length one between communicating LPs, a scenario could be generated requiring messages to be passed using two or three arcs on the Hypercube. This would in turn require the use of intermediate player-copies between the originating player and the destination player-copy to relay the information on to the next one in the sequence of LPs. Adjacent LPs would not pose a problem in this case since they are connected by a

single arc. The originating player can send a player message via SPECTRUM directly to the intended destination player-copy, circumventing the need for 'relay' player-copies.

Players can not be *within* the sensor range of another player when starting a scenario because players can only sense when other players enter (or exit) their sensor range. All benchmark scenarios are designed so that this situation does not occur.

VII. *Results, Conclusions, and Research Recommendations*

7.1 Introduction.

This chapter is a summary of the work accomplished on BATTLESIM during this research phase. Section 7.2 discusses what major accomplishments were achieved. Section 7.3 provides answers to the questions originally posed before research began. Lastly, Section 7.4 describes recommendations for areas in which further BATTLESIM research should be conducted.

7.2 Results.

This research effort began by identifying the basic requirements for BATTLESIM and determining what structures and methods were required to meet those requirements. The following list describes the *major* accomplishments which were achieved.

- Cleaned up existing code. This included interfacing Version 3 of TCHSIM into BATTLESIM in a modular manner, removing optimistic computation/local rollback, and updating the `player` object.
- Partitioned the battlefield into sectors; each one was responsible for all players within its boundaries, with each sector owned by a single LP for the duration of the scenario.
- Replaced full player state replication across all LPs with partial player state replication. Player state replication is now based on where the player is owned and what other sectors the player can sense into.
- Used a hierarchical, object-based approach to ensure that each structure in BATTLESIM is only aware of the information it needs to perform its task. This allowed BATTLESIM to gain the following software engineering-related benefits:

  1. Data Abstraction
  2. Information Hiding
  3. Modularity

4. Localization

- Implemented the `sector` and `playerset` objects along with their associated methods.
- Added the ability to use multiple scenario files, if desired.
- Mapped each sector in a scenario to an LP through TCHMAP.
- Modified scenarios to support multiple sectors per LP.
- Command-line arguments now allow the user to input small, frequently changed information in a manner which does not require recompilation of code or updating of data files.
- Used new data structures extensively to reduce the amount of wasted storage and remove the 1024 player limit each scenario could support.
- Implemented message-passing between sectors on the same node. Message passing between sectors on different nodes is simulated using print statements.

Message-passing between nodes was not completed because not enough time was left to do this in the manner desired (see research recommendation for inter-node message-passing). The use of print statements allow verification that the correct message is being sent, that its size is being correctly computed, and that the correct destination node is used.

The accomplishment of message-passing between nodes depended upon the completion of battlefield partitioning and message-passing between sectors on the same node first. Once implementation of the partitioned battlefield was well underway however, it became readily apparent that implementing the **`sector`**, **`playerset`**, and **`player`** objects in an object-based manner was going to be considerably more challenging than originally imagined. The number of methods necessary to access the objects without any knowledge of their underlying structures was significant, as Appendix B attests; BATTLESIM grew from a total of approximately 5000 lines of code to approximately 25,000 lines of code counting comments. Implementing handles as void pointers in the 'C' programming language was particulary vexing too. Nevertheless, the author felt that the extra time and effort spent

doing these tasks well now will result in more productive research with BATTLESIM in the future.

While shortcuts could have been taken throughout this effort to ensure timely completion of all objectives, my committee emphasized the need for *quality* code which exhibited the design philosophy characteristics listed in Chapter 3 *before quantity.* The importance of this personally became clear when my research phase "began" by untangling intermeshed software components within the last version of BATTLESIM. Suggestions for how to finish implementing dynamic partitioning are included under the recommendations section.

7.3 Conclusions.

Several questions, listed in Chapter 1, were posed at the outset of this research phase concerning sectors, players, and battlefield scenarios. This sections deals with answering those questions.

1. **What criteria should be used for deciding how big a battlefield sector is?**

   I propose that a heuristic such as "player density" i.e. the total number of players in the sector, be used in conjunction with a player weighting factor to determine when it is appropriate to change a sector's size. Each player's weighting factor could be based upon characteristics such as what kind of player it is, how fast it is moving, status (attacking, evading, or just moving), etc. The total player density/weighting factor in each sector could be measured — and each sector's size correspondingly adjusted — by BATTLESIM to ensure that the same computational and message-passing workload exists on each LP, to the degree possible.

   A 'window' could specify what range of weighting factors may exist on each LP without requiring a sector to be resized. If the LP's weighting factor increased (or decreased) outside this window, then a sector's size would be decreased (or increased) to change the number of players in it. To determine which boundary (or combination of boundaries) to change, the densities of all the sectors bordering the one in question can be checked, and the one(s) with the lowest weighting factor can have their sector boundary bordering the original sector modified.

Experiments should be run to determine what window size would help minimize "thrashing" between nodes i.e. the constant and undesired transfer of a player back and forth between two nodes. Thrashing, if it is serious, could lock up the simulation just like an infinite loop, so thorough analysis of window size for each scenario would be essential.

2. **How often should the sector size be changed?** The current version of BATTLESIM fixes sector size at initialization time and does not allow it to be changed, so experimentation with changing sector size during scenario execution was not possible. The faster the total workload on each LP changes, however, the more often it will be necessary to change battlefield sector sizes again to redistribute the LP workload evenly. The weighting factor could be used to determine when this is necessary based upon the 'window' of acceptable values mentioned in the previous quesion. When the LP's weighting factors stayed within its allowed window, no change in sector size would be performed. However, when the factor either increased or decreased outside the window, then the sector size would be decreased or increased, respectively.

3. **What shape should the sector be?** Four sector shapes were considered for implementation: the hexagon, square, strip and rectanglar cube. The rectanglar cube was chosen because it combined the best qualities of the three other structures. It is simple enough to formulate straightforward mathematical equations for boundary crossing, and yet it allows considerable flexibility in how a sector may dynamically change size.

4. **How and when should battlefield players be transferred from one sector to another?** Players are transferred when they either change which sector they reside in, or gain visibility into another sector. To accomplish this, a player message containing the entire contents of the source player should be built. Then TCHMAP is used to determine on which node the destination sector resides. If it is on the same node, then the player message is unpacked and the destination player immediately updated with its contents.

   If the destination sector resides on another node, then two messages are sent to the gaining node: the size of the player message and the message itself. The size is used

to determine the amount of space to allocate for the message on the destination node before it is received. Then the message is received, unpacked, and used to update the contents of the destination player.

5. **What kind of object should be used in the simulation to represent sectors?** A sector in BATTLESIM is a structure composed of several fields and lower-level structures. These fields and structures include:

   - sector ID
   - min and max x/y/z boundaries
   - a handle to a playerset
   - neighbors structure
   - next event time

   A "hierarchical approach" is used to implement the sector structure — the sector is composed of a playerset, which is composed of a player, which is composed of several attributes at the lowest level. The sector object, as well as the other major objects in BATTLESIM, utilize `methods` to support the insertion, update, and deletion of information. This object-based approach is purposely used to promote the principles of software engineering.

6. **How should battlefield scenarios be generated to test dynamic partitioning capabilities in BATTLESIM?** Once dynamic partitioning is implemented, two general kinds of scenarios should be generated: ones which start out with an even LP workload distribution and become uneven due to player clustering on the battlefield, and those which start out with an uneven LP workload distribution. BATTLESIM should be able to redistribute the workload in both cases. Tests should also check what size the weighting factor 'window' for each LP should be under various battlefield conditions. Some of the conditions which should be tested include:

   - player formations
   - number of players

- size of the players
- rate of change for each LP's total weighting factor
- effect of allowing 'windows' of different size on each LP

7.4 Research Recommendations.

7.4.1 Inter-Node Message Passing. Recommend that message-passing between nodes be implemented through the addition of a void pointer to the SPECTRUM message type — a pointer which references a data structure being passed between nodes. In the case of BATTLESIM, this pointer could first refer to the player message containing an entire player; it could be used later to build messages containing only those player fields which have been updated or just the next potential event associated with a given player.

A determination should then be made whether BATTLESIM speedup is achieved when a non-trivial parallel simulation is executed, and whether that speedup is then limited due to player next event determination. Hooks have already been added to BATTLESIM where messages should be sent.

7.4.2 Automated Scenario Generation Tools. Recommend that tools be created which allow scenario and map files to be generated in an easier fashion. These tools could interact directly with the user in a query/response manner.

Another approach is the application of AI techniques to generate portions of the scenario (like route, number of players, targets) ***automatically*** based upon information provided in a division-level standard military order.

7.4.3 Z Coordinate Axis Partitioning. Recommend that partitioning of the battlefield be extended into the z coordinate axis. The **neighbors structure** in the sector ADT already has 'hooks' in place to facilitate this upgrade. This would allow three dimensional partitioning to take place, and support experimentation with the concept of vertical "air corridors". Suggest that these air corriders then be integrated into an overall command and control structure, with lower-level corridors controlled by unit commanders and higher-level corridors controlled by division-level commanders or an AWACS unit.

7.4.4 Distributed Processing Environment. Recommend that BATTLESIM be modified to run in parallel using nodes on different computers, i.e. a distributed processing environment. This would allow experimentation with the data transmission protocol being developed for DIS, and help determine how to port BATTLESIM to future simulator platforms.

7.4.5 Varying Sector Sizes. Recommend that dynamic spatial partitioning be implemented to allow sector size to dynamically change; this should keep the workload associated with the simulation as evenly distributed among the scenario's LPs as possible. Some heuristic to control when and how much the sectors changed size — like the weighting factor 'window' described earlier — would have to be implemented as well.

7.4.6 Different Time Synchronization Protocols. Recommend that filters be developed under SPECTRUM to test the use of different time synchronization protocols. An optimistic filter should be developed which does not partially implement optimistic time synchronization like Soderholm did, but instead allows *all* nodes to execute optimistically and then roll back to a previous state using checkpoint information when necessary.

Saving checkpoint information would involve the periodic storage of all scenario state information. Since checkpoint datasets would be periodically discarded as well as saved, the user could not roll back any further than the earliest checkpoint still kept. To save a scenario's current state, base objects which support inheritance could be generated at the highest abstraction levels, with subordinate objects generated below them. This approach would allow scenario state information to be saved more easily at whatever level of abstraction is desired; a command to save a base object's state would automatically invoke methods to save the state of all its subordinate objects.

The author believe that dependencies between nodes due to player next event determination will require the use of an optimistic protocol in order to achieve significant speedup.

7.4.7 Interactive Control. Recommend further research be performed in developing an interface to control BATTLESIM interactively during execution. The user should be able to fully control execution through several parameters, including:

- Starting and stopping the simulation whenever desired
- Rewinding and fast-forwarding to any point in the scenario
- Zoom in and out of the battlefield
- Load any graphics file for display
- Moving players around on the battlefield at will

The interface should conform to the DIS protocol.

Appendix A. *A Benchmark Scenario Example*

A.1 Introduction.

While implementing spatial partitioning, player message passing and the other features new to this release of BATTLESIM, several new scenarios were created to ensure that these new features were operating correctly. These scenarios are "benchmarks" since they set the standards for what users can expect in terms of current BATTLESIM capabilities and ensure that future modifications do not adversely impact existing capabilities. This appendix lists one of the seven benchmark scenarios created to test the features added to BATTLESIM.

Benchmark scenario 13, a scenario designed to test the ability to manage a player crossing sector boundaries in both the x and y axes, is provided as the example. It consist of three subsections which describe the scenario's LP files, map file, and a diagram of the scenario executing. All scenario files follow the naming convention already described in Chapter 6.

A.2 Benchmark Scenario 13.

This scenario was designed to run with 8 LPs, so 8 scenario files are required to support it. However, since the scenario files for LPs 1 through 7 are identical, only one of them is shown. The required MAP file is next, followed by a diagram depicting the movements of all the players during the entire course of the scenario. This scenario was designed to ensure that a player could correctly cross sector boundaries in the +x, -x, +y, and -y directions in the same scenario *using all three boundary-crossing events.* The plane flies a zig-zag pattern to accomplish this.

A.2.1 Scenario Files.

```
****************************************************************************
* FILE: bench130.in4
* AUTHOR: Capt Ken Bergman
* REVISION: 1.0
* DATE: 14 Sep 92
* DESCRIPTION: This file contains 1 plane description. It is intended to
*              be used as one of many input files (this one for node 0).
*              The files, whether used together or individually, represent
*              an embarassingly parallel scenario designed to allow each
*              plane to fly its assigned routepoints with no detection, and
*              therefore no evasion or attack, of other aircraft. This file
*              was intended to be loaded on a single LP on the hypercube to
*              allow 1 object per node.
* HISTORY:
*   14 Sep 92 - Ver 1.0 Bergman
*         Original. Designed to make sure that aircraft are properly
*   replicated when passing from one sector to another on the
* SAME LP. In this particular benchmark scenario, one aircraft
* flies through sectors owned/controlled by LP 0. All other
* LP's (1-7) have no sectors or players assigned to them.
* This benchmark was specifically designed to check for proper
* scenario execution when players cross sector boundaries in
* the +x, -x, +y, and -y directions. The plane flies a zig-zag
* pattern in the x-axis first through all sectors, and then it
* flies a zig-zag pattern in the y-axis direction through all
* sectors.
****************************************************************************
* version number
V4.0
* terrain data filename
terrain.10
* terrain min coordinates (x, y, z)
0.1 0.1 0.1
* terrain max coordinates (x, y, z)
117000.0 118000.0 1000.0
* number of sectors (must be < 64)
8
* sector min/max boundaries (x,y,z values in order from 1st to last sectors)
```

```
0.1 59000. 0.1 29250. 118000. 1000.
29250. 59000. 0.1 58500. 118000. 1000.
58500. 59000. 0.1 87750. 118000. 1000.
87750. 59000. 0.1 117000. 118000. 1000.
0.1 0.1 0.1 29250. 59000. 1000.
29250. 0.1 0.1 58500. 59000. 1000.
58500. 0.1 0.1 87750. 59000. 1000.
87750. 0.1 0.1 117000. 59000. 1000.
* number of icon records
5
1 f18
2 mig1
3 missile
4 tank
5 truck
* object type thru max climb
1 1 1 0 1 1 1 1000 0 0 0 0 0 1 1 1 1 1 1
* number of route points
13
* route coordinates x,y,z (start to finish order)
8775. 110133.33 500.
108225. 110133.33 500.
108225. 7866.67 500.
8775. 7866.67 500.
8775. 102266.67 500.
20475. 102266.67 500.
20475. 19666.67 500.
43875. 19666.67 500.
43875. 102266.67 500.
73125. 102266.67 500.
73125. 19666.67 500.
96525. 19666.67 500.
96525. 102266.67 500.
* number of sensors
1
1 5850 1
* number of armaments
0
* armament descriptions (if above > 0)
* number of targets
3
* target descriptions (if above > 0)
1 0 0 0
4 0 0 0
5 0 0 0
* number of defensive systems
0
* defensive system descriptions (if above > 0)
* END OF OBJECT
```

```
********************************************************************************
* FILE: bench131.in4
* AUTHOR: Capt Ken Bergman
* REVISION: 1.0
* DATE: 14 Sep 92
* DESCRIPTION: This file contains 1 plane description. It is intended to
*              be used as one of many input files (this one for node 1).
*              The files, whether used together or individually, represent
*              an embarassingly parallel scenario designed to allow each
*              plane to fly its assigned routepoints with no detection, and
*              therefore no evasion or attack, of other aircraft. This file
*              was intended to be loaded on a single LP on the hypercube to
*              allow 1 object per node.
* HISTORY:
*   14 Sep 92 - Ver 1.0 Bergman
*         Original. Designed to make sure that aircraft are properly
*   replicated when passing from one sector to another on the
* SAME LP. In this particular benchmark scenario, one aircraft
* flies through sectors owned/controlled by LP 0. All other
* LP's (1-7) have no sectors or players assigned to them.
* This benchmark was specifically designed to check for proper
* scenario execution when players cross sector boundaries in
* the +x, -x, +y, and -y directions. The plane flies a zig-zag
* pattern in the x-axis first through all sectors, and then it
* flies a zig-zag pattern in the y-axis direction through all
* sectors.
********************************************************************************
* version number
V4.0
* terrain data filename
terrain.10
* terrain min coordinates (x, y, z)
0.1 0.1 0.1
* terrain max coordinates (x, y, z)
117000.0 118000.0 1000.0
* number of sectors (must be < 64)
8
* sector min/max boundaries (x,y,z values in order from 1st to last sectors)
0.1 59000. 0.1 29250. 118000. 1000.
29250. 59000. 0.1 58500. 118000. 1000.
58500. 59000. 0.1 87750. 118000. 1000.
87750. 59000. 0.1 117000. 118000. 1000.
0.1 0.1 0.1 29250. 59000. 1000.
29250. 0.1 0.1 58500. 59000. 1000.
58500. 0.1 0.1 87750. 59000. 1000.
87750. 0.1 0.1 117000. 59000. 1000.
* number of icon records
5
1 f18
2 mig1
3 missile
4 tank
5 truck
* END OF OBJECT
```

A.2.2 Map File.

```
********************************************************************************
* FILE: battlesim13.map
* AUTHOR: Capt Ken Bergman
* REVISION: 1.0
* DATE: 14 Sep 92
* DESCRIPTION:  This file contains the battlesim sector-to-LP description
* for benchmark scenario 13. The BATTLESIM application must
* be executed with 8 LPs to use this particular map file.
*               Each line describes a single mapping from a single sector ID
* to a single LP id, in that order.  There should be exactly
* as many lines as there are sectors from the scenario file(s)
* being used for a given simulation run.
* HISTORY:
*   14 Sep 92 - Ver 1.0 Bergman
* Created this original mapping file.
********************************************************************************
* Mapping sector 1 to LP 0
1 0
* Mapping sector 2 to LP 0
2 0
* Mapping sector 3 to LP 0
3 0
* Mapping sector 4 to LP 0
4 0
* Mapping sector 5 to LP 0
5 0
* Mapping sector 6 to LP 0
6 0
* Mapping sector 7 to LP 0
7 0
* Mapping sector 8 to LP 0
8 0
```

A.2.3 Sccnario Diagram. This diagram depicts the movement of Benchmark 13's player throughout the course of the scenario.

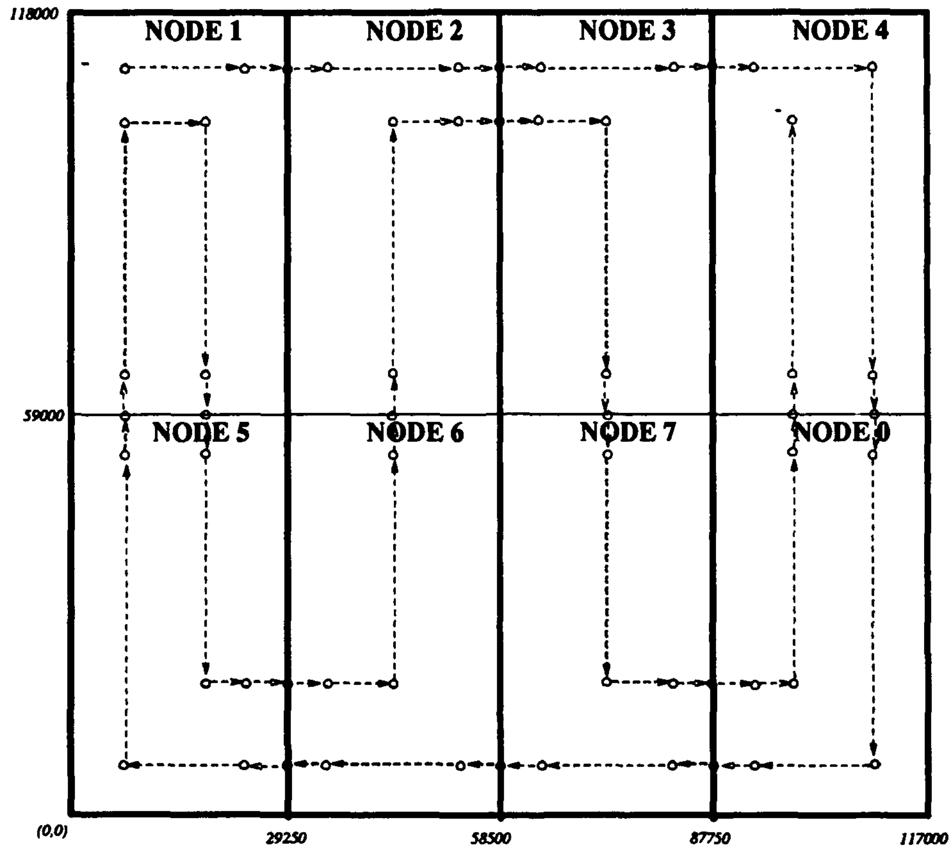

Figure 18. Benchmark Scenario 13

Appendix B. *Major BATTLESIM Methods*

B.1 Introduction.

This appendix lists the methods developed to access, modify and delete all fields and structures within the player, playerset, and sector objects in BATTLESIM. These object-based methods were designed using well-established principles of software engineering including:

- Data Abstraction
- Information Hiding
- Modularity
- Localization

Data abstraction is the ability to view information from several different perspectives, using whichever one is most convenient at the time (1:32). BATTLESIM applies the concept of data abstraction through the formation of "ladders of abstraction" in which each higher level of abstraction is built from lower levels. In BATTLESIM, some of the main levels of abstraction are the player, playerset, and sector described in this appendix. Each of these levels identifies an abstract data type (ADT), with each characterized by a set of state information and a set of methods applicable to each object instantiated from that abstact data type.

While data abstraction is the extraction of essential details at a given level, the purpose of information hiding is to make inaccessible certain details that should not affect other parts of a system(31:67). Information hiding suppresses how an object is implemented, making the user focus his attention on the higher-level abstraction where it belongs. Structures in BATTLESIM implement information hiding through the use of static variable and structure declarations, and the use of void pointers whenever possible.

Modularity deals with how the structure of an object can make the attainment of some purpose easier; it is 'purposeful structuring'(31:67). With each year of research and new software development, BATTLESIM is becoming an increasingly complex battlefield

simulation which combines the software design methodologies of top-down and bottom-up design. It is top-down because some elements of BATTLESIM (like SPECTRUM) employ a top-down "layered approach" in which higher-level modules relate more closely to higher-level abstractions; the higher-level functions specify *what* to do, while the lower-level ones say *how* to do it. It is bottom-up because it employs reuseable software components like TCHMAP and Rizza's doubly linked list package which are combined at the lower levels of BATTLESIM to form an integrated simulation platform.

Localization deals with the physical location of data structures, program modules, and other elements in a software package. Ideally, related resources are grouped together in one physical "module", typically a file. In BATTLESIM, localization is accomplished by combining abstract data types along with their associated state information and methods into distinct files which are easily identified by their naming conventions, like `player.c`, `playerset.c`, `sector.c`, `sensor.c`, and `message.c`.

All of these principles lend themselves toward striving to meet the goals of software engineering, just some of which include:

- understandability
- reliability
- modifiability
- efficiency

Shorthand naming conventions have been developed to help enhance reader understanding and program clarity. These conventions are mainly used within the BATTLESIM method names and arguments described in the following sections; however, they are used to a lesser degree within the method bodies and comments too. The naming conventions include:

- P - designates a *player* in the simulation
- Pset - designates a *playerset* in the simulation
- Pset_ptr - designates a *pointer to a playerset* in the simulation

- S - designates a *sector* in the simulation battlefield
- S_id - represent a *sector identifier*, an integer value unique to each sector in the simulation battlefield
- P_id - designates a *player identifier*, an integer value unique to each player which is owned by a sector in the simulation (not just a copy)
- ll - designates a *linked list*

B.2 Methods for Accessing Player Object.

This section lists all of the object-based methods developed to access, modify, and delete the fields and structures within the player ADT. All of the methods, as well as the player structure itself, are encapsulated within the file `player.c`. This file is comprised of 5863 lines of code in 129 methods.

1. **Function Name/Parameters:** det_route_event(player)

   Parameter Types: struct object_attributes *player

   Description: Determines the next route point event for the player and builds the corresponding event. If none exists, then it returns a NULL value.

   Return Value: void * to next route point event

2. **Function Name/Parameters:** max_sensor_range(player)

   Parameter Types: struct object_attributes *player

   Description: This function is used by sensor_check to determine the range of the sensor being used. If no sensor exists for the specified player, then it returns a value of 0.

   Return Value: int

3. **Function Name/Parameters:** list_player(player)

   Parameter Types: struct object_attributes *player

   Description: Lists all the fields of a specified player. USED FOR DEBUGGING.

   Return Value: none

4. **Function Name/Parameters:** getPsector_id(player)

   Parameter Types: struct object_attributes *player

   Description: Retrieves a particular player's sector_id value.

   Return Value: int

5. **Function Name/Parameters:** setPsector_id(player, new_sector_id)

   Parameter Types: struct object_attributes *player, int new_sector_id

   Description: Sets a particular player's sector_id value.

   Return Value: none

6. **Function Name/Parameters:** getPown_player(player)

   Parameter Types: struct object_attributes *player

   Description: Retrieves a particular player's own_player value.

   Return Value: int

7. **Function Name/Parameters:** setPown_player(player, own_this_player)

   Parameter Types:struct object_attributes *player, int own_this_player

   Description: Sets a particular player's own_player value.

   Return Value: none

8. **Function Name/Parameters:** getPsize(player)

   Parameter Types: struct object_attributes *player

   Description: Retrieves a particular player's player_size value.

   Return Value: int

9. **Function Name/Parameters:** setPsize(player, player_size)

   Parameter Types: struct object_attributes *player, int player_size

   Description: Sets a particular player's player_size value.

   Return Value: none

10. **Function Name/Parameters:** getPobj_type(player)

    Parameter Types: struct object_attributes *player

    Description: Retrieves a particular player's object_type value.

    Return Value: int

11. **Function Name/Parameters:** setPobj_type(player, obj_type)

    Parameter Types: struct object_attributes *player, int obj_type

    Description: Sets a particular player's object_type value.

    Return Value: none

12. **Function Name/Parameters:** getPobj_id(player)

    Parameter Types: struct object_attributes *player

    Description: Retrieves a particular player's object_id value.

    Return Value: int

13. **Function Name/Parameters:** setPobj_id(player, new_obj_id)
    Parameter Types: struct object_attributes *player, int new_obj_id
    Description: Sets a particular player's object_id value.
    Return Value: none
14. **Function Name/Parameters:** getPobj_lylty(player)
    Parameter Types: struct object_attributes *player
    Description: Retrieves a particular player's object_loyalty value.
    Return Value: int
15. **Function Name/Parameters:** setPobj_lylty(player, obj_lylty)
    Parameter Types: struct object_attributes *player, int obj_lylty
    Description: Sets a particular player's object_loyalty value.
    Return Value: none
16. **Function Name/Parameters:** getPcurr_time(player)
    Parameter Types: struct object_attributes *player
    Description: Retrieves a particular player's current_time value.
    Return Value: double
17. **Function Name/Parameters:** setPcurr_time(player, cur_tim)
    Parameter Types: struct object_attributes *player, double cur_tim
    Description: Sets a particular player's current_time value.
    Return Value: none
18. **Function Name/Parameters:** getPfuel_stat(player)
    Parameter Types: struct object_attributes *player
    Description: Retrieves a particular player's fuel_status value.
    Return Value: int
19. **Function Name/Parameters:** setPfuel_stat(player, fuel_stat)
    Parameter Types: struct object_attributes *player, int fuel_stat
    Description: Sets a particular player's fuel_status value.
    Return Value: none
20. **Function Name/Parameters:** getPcond(player)
    Parameter Types: struct object_attributes *player
    Description: Retrieves a particular player's condition value.
    Return Value: int

21. **Function Name/Parameters:** setPcond(player, cond)

    Parameter Types: struct object_attributes *player, int cond

    Description: Sets a particular player's condition value.

    Return Value: none

22. **Function Name/Parameters:** getPvul(player)

    Parameter Types: struct object_attributes *player

    Description: Retrieves a particular player's vulnerability value.

    Return Value: int

23. **Function Name/Parameters:** setPvul(player, vulnblty)

    Parameter Types: struct object_attributes *player, int vulnblty

    Description: Sets a particular player's vulnerability value.

    Return Value: none

24. **Function Name/Parameters:** getPloc(player)

    Parameter Types: struct object_attributes *player

    Description: Retrieves a particular player's location (in x, y, and z coordinates).

    Return Value: struct location_type

25. **Function Name/Parameters:** getPloc_x(player)

    Parameter Types: struct object_attributes *player

    Description: Retrieves a particular player's location (in x coordinate only).

    Return Value: double

26. **Function Name/Parameters:** getPloc_y(player)

    Parameter Types: struct object_attributes *player

    Description: Retrieves a particular player's location (in y coordinate only).

    Return Value: double

27. **Function Name/Parameters:** getPloc_z(player)

    Parameter Types: struct object_attributes *player

    Description: Retrieves a particular player's location (in z coordinate only).

    Return Value: double

28. **Function Name/Parameters:** setPloc(player, x_value, y_value, z_value)

    Parameter Types: struct object_attributes *player, double x_value, double y_value, double z_value

    Description: Sets a particular player's location (in x, y, and z coordinates).

    Return Value: none

29. **Function Name/Parameters:** getPvel(player)

    Parameter Types: struct object_attributes *player

    Description: Retrieves a particular player's velocity (in x, y, and z components).

    Return Value: struct xyz_velocities

30. **Function Name/Parameters:** getPvel_x(player)

    Parameter Types: struct object_attributes *player

    Description: Retrieves a particular player's velocity (in x component only).

    Return Value: double

31. **Function Name/Parameters:** getPvel_y(player)

    Parameter Types: struct object_attributes *player

    Description: Retrieves a particular player's velocity (in y component only).

    Return Value: double

32. **Function Name/Parameters:** getPvel_z(player)

    Parameter Types: struct object_attributes *player

    Description: Retrieves a particular player's velocity (in z component only).

    Return Value: double

33. **Function Name/Parameters:** setPvel_x(player)

    Parameter Types: struct object_attributes *player

    Description: Sets a particular player's velocity (in x component only).

    Return Value: none

34. **Function Name/Parameters:** setPvel_y(player)

    Parameter Types: struct object_attributes *player

    Description: Sets a particular player's velocity (in y component only).

    Return Value: none

35. **Function Name/Parameters:** setPvel_z(player)

    Parameter Types: struct object_attributes *player

    Description: Sets a particular player's velocity (in z component only).

    Return Value: none

36. **Function Name/Parameters:** setPvel(player, x_value, y_value, z_value))

    Parameter Types: struct object_attributes *player, double x_value, double y_value, double z_value

    Description: Sets a particular player's velocity (in x, y, and z components).

    Return Value: none

37. **Function Name/Parameters:** getPorien(player)

    Parameter Types: struct object_attributes *player

    Description: Retrieves a particular player's orientation (in roll, pitch, and yaw values).

    Return Value: struct orientation_type

38. **Function Name/Parameters:** getPorien_roll(player)

    Parameter Types: struct object_attributes *player

    Description: Retrieves a particular player's orientation (roll value only).

    Return Value: double

39. **Function Name/Parameters:** getPorien_pitch(player)

    Parameter Types: struct object_attributes *player

    Description: Retrieves a particular player's orientation (pitch value only).

    Return Value: double

40. **Function Name/Parameters:** getPorien_yaw(player)

    Parameter Types: struct object_attributes *player

    Description: Retrieves a particular player's orientation (yaw value only).

    Return Value: double

41. **Function Name/Parameters:** setPorien(player, roll_value, pitch_value, yaw_value)

    Parameter Types: struct object_attributes *player, double roll_value, double pitch_value, double yaw_value

    Description: Sets a particular player's orientation (roll, pitch, and yaw values).

    Return Value: none

42. **Function Name/Parameters:** getProt(player)

    Parameter Types: struct object_attributes *player

    Description: Retrieves a particular player's rotation rate (roll, pitch, and yaw values).

    Return Value: struct rotation_rates

43. **Function Name/Parameters:** getProt_roll(player)

    Parameter Types: struct object_attributes *player

    Description: Retrieves a particular player's rotation rate (roll value only).

    Return Value: double

44. **Function Name/Parameters:** getProt_pitch(player)

    Parameter Types: struct object_attributes *player

    Description: Retrieves a particular player's rotation rate (pitch value only).

    Return Value: double

45. **Function Name/Parameters:** getProt_yaw(player)

    Parameter Types: struct object_attributes *player

    Description: Retrieves a particular player's rotation rate (yaw value only).

    Return Value: double

46. **Function Name/Parameters:** setProt(player, roll_value, pitch_value, yaw_value)

    Parameter Types: struct object_attributes *player, double roll_value, double pitch_value, double yaw_value

    Description: Sets a particular player's rotation rates (roll, pitch, and yaw values).

    Return Value: none

47. **Function Name/Parameters:** getPoper(player)

    Parameter Types: struct object_attributes *player

    Description: Retrieves a particular player's operator information (experience and threat_knowledge).

    Return Value: struct operator_type

48. **Function Name/Parameters:** getPoper_exp(player)

    Parameter Types: struct object_attributes *player

    Description: Retrieves a particular player's operator information (experience only).

    Return Value: int

49. **Function Name/Parameters:** getPoper_tknow(player)

    Parameter Types: struct object_attributes *player

    Description: Retrieves a particular player's operator information (threat_knowledge only).

    Return Value: int

50. **Function Name/Parameters:** setPoper(player, expernce, t_know)

    Parameter Types: struct object_attributes *player, int expernce, int t_know

    Description: Sets a particular player's operator information (experience and threat knowledge).

    Return Value: none

51. **Function Name/Parameters:** getPperfchar(player)

    Parameter Types: struct object_attributes *player

    Description: Retrieves a particular player's performance characteristics. This includes:

    - min_turn_radius
    - max_speed

- ave_fuel_cons_rate
- max_climb_rate

Return Value: struct performance_characteristics

52. **Function Name/Parameters:** getPperfchar_mtr(player)

    Parameter Types: struct object_attributes *player

    Description: Retrieves a particular player's performance characteristics (min_turn_radius only).

    Return Value: int

53. **Function Name/Parameters:** getPperfchar_ms(player)

    Parameter Types: struct object_attributes *player

    Description: Retrieves a particular player's performance characteristics (max_speed only).

    Return Value: int

54. **Function Name/Parameters:** getPperfchar_afcr(player)

    Parameter Types: struct object_attributes *player

    Description: Retrieves a particular player's performance characteristics (ave_fuel_cons_rate only).

    Return Value: int

55. **Function Name/Parameters:** getPperfchar_mcr(player)

    Parameter Types: struct object_attributes *player

    Description: Retrieves a particular player's performance characteristics (max_climb_rate only).

    Return Value: int

56. **Function Name/Parameters:** setPperfchar(player, mtr, ms, afcr, mcr)

    Parameter Types: struct object_attributes *player, int mtr, int ms, int afcr, int mcr

    Description: Sets a particular player's performance characteristics. This includes:

    - min_turn_radius
    - max_speed
    - ave_fuel_cons_rate
    - max_climb_rate

    Return Value: none

57. **Function Name/Parameters:** getPcopies_ll(player)

    Parameter Types: struct object_attributes *player

    Description: Retrieves a particular player's linked list of player copies.

    Return Value: void * to linked list

58. **Function Name/Parameters:** getPcopies_first(player)

    Parameter Types: struct object_attributes *player

    Description: Retrieves a particular player's FIRST player-copy in its linked list of player-copies.

    Return Value: void * to player

59. **Function Name/Parameters:** addPcopies_ll(player)

    Parameter Types: struct object_attributes *player

    Description: Adds a linked list of player-copies to a specified player. This function assumes that no linked list of player-copies already exists for the specified player.

    Return Value: none

60. **Function Name/Parameters:** freePcopies_ll(player)

    Parameter Types: struct object_attributes *player

    Description: Frees the memory associated with the player-copies in a specified player's Pcopies_ll.

    Return Value: none

61. **Function Name/Parameters:** addPnewcopies_ll(player)

    Parameter Types: struct object_attributes *player

    Description: Adds a linked list of player-copies to a specified player. This function assumes that an EMPTY linked list of player-copies already exists for the specified player.

    Return Value: none

62. **Function Name/Parameters:** addPcopy(player, S_id_ptr)

    Parameter Types: struct object_attributes *player, int *S_id_ptr

    Description: Adds another player-copy to a specified player's player-copies linked list.

    Return Value: none

63. **Function Name/Parameters:** delPcopy(player, S_id_ptr)

    Parameter Types: struct object_attributes *player, int *S_id_ptr

    Description: Deletes a specified player-copy from a player's player-copies linked list.

    Return Value: none

64. **Function Name/Parameters:** makePcopies_ll(player)

    Parameter Types: struct object_attributes *player

    Description: Generates a new, empty player-copies linked list for a given player.

    Return Value: none

65. **Function Name/Parameters:** getProute_ll(player)

    Parameter Types: struct object_attributes *player

    Description: Retrieves a particular player's linked list of route points.

    Return Value: void * to linked list

66. **Function Name/Parameters:** getProute_firstpt(player)

    Parameter Types: struct object_attributes *player

    Description: Retrieves a particular player's first point on its linked list of route points.

    Return Value: void * to route point

67. **Function Name/Parameters:** getProute_firstpt_x(player)

    Parameter Types: struct object_attributes *player

    Description: Retrieves a particular player's first point (x value only) on its linked list of route points.

    Return Value: double

68. **Function Name/Parameters:** getProute_firstpt_y(player)

    Parameter Types: struct object_attributes *player

    Description: Retrieves a particular player's first point (y value only) on its linked list of route points.

    Return Value: double

69. **Function Name/Parameters:** getProute_firstpt_z(player)

    Parameter Types: struct object_attributes *player

    Description: Retrieves a particular player's first point (z value only) on its linked list of route points.

    Return Value: double

70. **Function Name/Parameters:** getPpt_x(pt)

    Parameter Types: struct location_type *pt

    Description: Retrieves the x-value of a provided point (may NOT be the first point in a player's linked list of route points).

    Return Value: double

71. **Function Name/Parameters:** getPpt_y(pt)

    Parameter Types: struct location_type *pt

    Description: Retrieves the y-value of a provided point (may NOT be the first point in a player's linked list of route points).

    Return Value: double

72. **Function Name/Parameters:** getPpt_z(pt)

    Parameter Types: struct location_type *pt

    Description: Retrieves the z-value of a provided point (may NOT be the first point in a player's linked list of route points).

    Return Value: double

73. **Function Name/Parameters:** addProute_ll(player, route_ll)

    Parameter Types: struct object_attributes *player, void *route_ll

    Description: Adds a linked list of route points to a specified player. This function assumes that no linked list of route points already exists for the specified player.

    Return Value: none

74. **Function Name/Parameters:** freeProute_ll(player, route_ll)

    Parameter Types: struct object_attributes *player, void *route_ll

    Description: Frees the memory associated with the route-pts in a specified player's route_ll.

    Return Value: none

75. **Function Name/Parameters:** addPnewroute_ll(player, route_ll)

    Parameter Types: struct object_attributes *player, void *route_ll

    Description: Adds a linked list of route points to a specified player. This function assumes that an EMPTY linked list of route points already exists for the specified player. Therefore this function performs the same functions as addProute_ll, except that it does NOT generate a new linked list.

    Return Value: none

76. **Function Name/Parameters:** addProute_pt(player, x_value, y_value, z_value)

    Parameter Types: struct object_attributes *player, double x_value, double y_value, double z_value

    Description: Adds another route point to a player's route_data linked list.

    Return Value: none

77. **Function Name/Parameters:** getPsensor_ll(player)

    Parameter Types: struct object_attributes *player

    Description: Retrieves a particular player's linked list of sensors.

    Return Value: void * to linked list

78. **Function Name/Parameters:** getPsensors_first(player)

    Parameter Types: struct object_attributes *player

    Description: Retrieves a particular player's first sensor on its linked list of sensors.

    Return Value: void * to sensor

79. **Function Name/Parameters:** getPsensors_first_type(player)

    Parameter Types: struct object_attributes *player

    Description: Retrieves a particular player's first sensor (type only) on its linked list of sensors.

    Return Value: int

80. **Function Name/Parameters:** getPsensors_first_range(player)

    Parameter Types: struct object_attributes *player

    Description: Retrieves a particular player's first sensor (range only) on its linked list of sensors.

    Return Value: int

81. **Function Name/Parameters:** getPsensors_first_res(player)

    Parameter Types: struct object_attributes *player

    Description: Retrieves a particular player's first sensor (resolution only) on its linked list of sensors.

    Return Value: int

82. **Function Name/Parameters:** getPsensor_type(sensor)

    Parameter Types: struct sensors *sensor

    Description: Retrieves the type value of a provided sensor (may NOT be the first sensor in a player's linked list of sensors).

    Return Value: int

83. **Function Name/Parameters:** getPsensor_range(sensor)

    Parameter Types: struct sensors *sensor

    Description: Retrieves the range value of a provided sensor (may NOT be the first sensor in a player's linked list of sensors).

    Return Value: int

84. **Function Name/Parameters:** getPsensor_res(sensor)

    Parameter Types: struct sensors *sensor

    Description: Retrieves the resolution value of a provided sensor (may NOT be the first sensor in a player's linked list of sensors).

    Return Value: int

85. **Function Name/Parameters:** addPsensor_ll(player, sensor_ll)

    Parameter Types: struct object_attributes *player, void *sensor_ll

    Description: Adds a linked list of sensors to a specified player. This function assumes that no linked list of sensors already exists for the specified player.

    Return Value: none

86. **Function Name/Parameters:** freePsensor_ll(player, sensor_ll)

    Parameter Types: struct object_attributes *player, void *sensor_ll

    Description: Frees the memory associated with the sensors in a specified player's sensor_ll.

    Return Value: none

87. **Function Name/Parameters:** addPsensor(player, type_value, range_value, resolution_value)

    Parameter Types: struct object_attributes *player, int type_value, int range_value, int resolution_value

    Description: Adds another sensor to a player's sensors linked list.

    Return Value: none

88. **Function Name/Parameters:** getParms_ll(player)

    Parameter Types: struct object_attributes *player

    Description: Retrieves a particular player's linked list of armaments.

    Return Value: void * to linked list

89. **Function Name/Parameters:** getParms_first(player)

    Parameter Types: struct object_attributes *player

    Description: Retrieves a particular player's first armament on its linked list of armaments.

    Return Value: void * to armament

90. **Function Name/Parameters:** getParms_first_type(player)

    Parameter Types: struct object_attributes *player

    Description: Retrieves a particular player's first armament (type only) on its linked list of armaments.

    Return Value: int

91. **Function Name/Parameters:** getParms_first_range(player)

    Parameter Types: struct object_attributes *player

    Description: Retrieves a particular player's first armament (range only) on its linked list of armaments.

    Return Value: int

92. **Function Name/Parameters:** getParms_first_lethality(player)

    Parameter Types: struct object_attributes *player

    Description: Retrieves a particular player's first armament (lethality only) on its linked list of armaments.

    Return Value: int

93. **Function Name/Parameters:** getParms_first_accuracy(player)

    Parameter Types: struct object_attributes *player

    Description: Retrieves a particular player's first armament (accuracy only) on its linked list of armaments.

    Return Value: int

94. **Function Name/Parameters:** getParms_first_speed(player)

    Parameter Types: struct object_attributes *player

    Description: Retrieves a particular player's first armament (speed only) on its linked list of armaments.

    Return Value: int

95. **Function Name/Parameters:** getParms_first_count(player)

    Parameter Types: struct object_attributes *player

    Description: Retrieves a particular player's first armament (count only) on its linked list of armaments.

    Return Value: int

96. **Function Name/Parameters:** getParm_type(player)

    Parameter Types: struct object_attributes *player

    Description: Retrieves the type value of a provided armament (may NOT be the first armament in a player's linked list.

    Return Value: int

97. **Function Name/Parameters:** getParm_range(player)

    Parameter Types: struct object_attributes *player

    Description: Retrieves the range value of a provided armament (may NOT be the first armament in a player's linked list.

    Return Value: int

98. **Function Name/Parameters:** getParm_lethality(player)

    Parameter Types: struct object_attributes *player

    Description: Retrieves the lethality value of a provided armament (may NOT be the first armament in a player's linked list.

    Return Value: int

99. **Function Name/Parameters:** getParm_accuracy(player)

    Parameter Types: struct object_attributes *player

    Description: Retrieves the accuracy value of a provided armament (may NOT be the first armament in a player's linked list.

    Return Value: int

100. **Function Name/Parameters:** getParm_speed(player)

    Parameter Types: struct object_attributes *player

    Description: Retrieves the speed value of a provided armament (may NOT be the first armament in a player's linked list.

    Return Value: int

101. **Function Name/Parameters:** getParm_count(player)

    Parameter Types: struct object_attributes *player

    Description: Retrieves the count value of a provided armament (may NOT be the first armament in a player's linked list.

    Return Value: int

102. **Function Name/Parameters:** addParms_ll(player, arms_ll)

    Parameter Types: struct object_attributes *player, void *arms_ll

    Description: Adds a linked list of armaments to a specified player. This function assumes that no linked list of armaments already exists for the specified player.

    Return Value: none

103. **Function Name/Parameters:** freeParms_ll(player, arms_ll)

    Parameter Types: struct object_attributes *player, void *arms_ll

    Description: Frees the memory associated with the armaments in a specified player's armaments_ll.

    Return Value: none

104. **Function Name/Parameters:** addParmament(player, typ, rng, lethal, accur, spd, cnt)

    Parameter Types: struct object_attributes *player, int typ, int rng, int lethal, int accur, int spd, int cnt

    Description: Adds another armament to a player's armaments linked list.

    Return Value: none

105. **Function Name/Parameters:** getPdefense_ll(player)

    Parameter Types: struct object_attributes *player

    Description: Retrieves a particular player's linked list of defensive systems.

    Return Value: void * to linked list

106. **Function Name/Parameters:** getPdefense_first(player)

    Parameter Types: struct object_attributes *player

    Description: Retrieves a particular player's first defensive system on its linked list of defensive systems.

    Return Value: void * to defensive system

107. **Function Name/Parameters:** getPdefense_first_type(player)

    Parameter Types: struct object_attributes *player

    Description: Retrieves a particular player's first defensive system (type only) on its linked list of defensive systems.

    Return Value: int

108. **Function Name/Parameters:** getPdefense_first_range(player)

    Parameter Types: struct object_attributes *player

    Description: Retrieves a particular player's first defensive system (range only) on its linked list of defensive systems.

    Return Value: int

109. **Function Name/Parameters:** getPdefense_first_effect(player)

    Parameter Types: struct object_attributes *player

    Description: Retrieves a particular player's first defensive system (effectiveness only) on its linked list of defensive systems.

    Return Value: int

110. **Function Name/Parameters:** getPdefense_type(defense)

    Parameter Types: struct defensive_systems defense

    Description: Retrieves the type value of a provided defensive system (may NOT be the first defensive system in a player's linked list of defensive systems).

    Return Value: int

111. **Function Name/Parameters:** getPdefense_range(defense)

    Parameter Types: struct defensive_systems defense

    Description: Retrieves the range value of a provided defensive system (may NOT be the first defensive system in a player's linked list of defensive systems).

    Return Value: int

112. **Function Name/Parameters:** getPdefense_effect(defense)

    Parameter Types: struct defensive_systems defense

    Description: Retrieves the effectiveness value of a provided defensive system (may NOT be the first defensive system in a player's linked list of defensive systems).

    Return Value: int

113. **Function Name/Parameters:** addPdefense_ll(player, defense_ll)

Parameter Types: struct object_attributes *player, void *defense_ll

Description: Adds a linked list of defensive systems to a specified player. This function assumes that no linked list of defensive systems already exists for the specified player.

Return Value: none

114. **Function Name/Parameters:** freePdefense_ll(player, defense_ll)

Parameter Types: struct object_attributes *player, void *defense_ll

Description: Frees the memory associated with the defensive_systems in a specified player's defense_ll.

Return Value: none

115. **Function Name/Parameters:** addPdefense(player, typ, rng, effect)

Parameter Types: struct object_attributes *player, int typ, int rng, int effect

Description: Adds another defensive system to a player's defensive system linked list.

Return Value: none

116. **Function Name/Parameters:** getPtarglist_ll(player)

Parameter Types: struct object_attributes *player

Description: Retrieves a particular player's linked list of targets.

Return Value: void * to linked list

117. **Function Name/Parameters:** getPtarglist_first(player)

Parameter Types: struct object_attributes *player

Description: Retrieves a particular player's first target on its linked list of targets.

Return Value: void * to target

118. **Function Name/Parameters:** getPtarglist_type(player)

Parameter Types: struct object_attributes *player

Description: Retrieves a particular player's first target (type only) on its linked list of defensive systems.

Return Value: int

119. **Function Name/Parameters:** getPtarglist_loc(player)

Parameter Types: struct object_attributes *player

Description: Retrieves a particular player's first target (location only) on its linked list of targets.

Return Value: struct location_type

120. **Function Name/Parameters:** getPtarglist_loc_x(player)

 Parameter Types: struct object_attributes *player

 Description: Retrieves a particular player's first target (x location coordinate only) on its linked list of targets.

 Return Value: double

121. **Function Name/Parameters:** getPtarglist_loc_y(player)

 Parameter Types: struct object_attributes *player

 Description: Retrieves a particular player's first target (y location coordinate only) on its linked list of targets.

 Return Value: double

122. **Function Name/Parameters:** getPtarglist_loc_z(player)

 Parameter Types: struct object_attributes *player

 Description: Retrieves a particular player's first target (z location coordinate only) on its linked list of targets.

 Return Value: double

123. **Function Name/Parameters:** getPtarget_type(target)

 Parameter Types: struct targets *target

 Description: Retrieves the type value of a specified target (may NOT be the first target in a player's linked list).

 Return Value: int

124. **Function Name/Parameters:** getPtarget_loc_x(target)

 Parameter Types: struct targets *target

 Description: Retrieves the location (x-value only) of a target (may NOT be the first target in a player's linked list).

 Return Value: double

125. **Function Name/Parameters:** getPtarget_loc_y(target)

 Parameter Types: struct targets *target

 Description: Retrieves the location (y-value only) of a target (may NOT be the first target in a player's linked list).

 Return Value: double

126. **Function Name/Parameters:** getPtarget_loc_z(target)

 Parameter Types: struct targets *target

 Description: Retrieves the location (z-value only) of a target (may NOT be the first target in a player's linked list).

 Return Value: double

127. **Function Name/Parameters:** addPtarglist_ll(player, targlist_ll)

    Parameter Types: struct object_attributes *player, void *targlist_ll

    Description: Adds a linked list of targets to a specified player. This function assumes that no linked list of targets already exists for the specified player.

    Return Value: none

128. **Function Name/Parameters:** freePtarglist_ll(player, targlist_ll)

    Parameter Types: struct object_attributes *player, void *targlist_ll

    Description: Frees the memory associated with the target_list in a specified player's targlist_ll.

    Return Value: none

129. **Function Name/Parameters:** addPtarget(player, typ, x_value, y_value, z_value)

    Parameter Types: struct object_attributes *player, int x_value, int y_value, int z_value

    Description: Adds another target to a player's target linked list.

    Return Value: none

B.3 Methods for Accessing Playerset Object.

This section lists all of the object-based methods developed to access, modify, and delete the fields and structures within the playerset ADT. All of the methods, as well as the playerset structure itself, are encapsulated within the file **`playerset.c`**. This file is comprised of 1164 lines of code in 20 methods.

1. **Function Name/Parameters:** list_Pset(Pset_ptr)

    Parameter Types: void *Pset_ptr

    Description: Lists all the players contained within a specified playerset, including their respective fields. USED FOR DEBUGGING.

    Return Value: none

2. **Function Name/Parameters:** copy_Pset_player(P_id1, P_id2)

    Parameter Types: struct object_attributes *P_id1, *P_id2

    Description: Copy a player from one playerset to another one. This is done by calling the low-level get routines for each of the fields within the old player's structure, and calling the low-level set routines for each of the fields within the new player's structure. This will prevent errors that would inevitably result from just copying the pointer to the original player's structure.

    Return Value: none

3. **Function Name/Parameters:** get_hash_entry(Pset_ptr, index)

   Parameter Types: void *Pset_ptr[], int index

   Description: Retrieves the correct playerset array entry (corresponding to a hash table entry). This entry is a linked list, even though it is being passed back as a void pointer. If there are no players to be retrieved or the function call was not made to a valid linked list, then the function returns a NULL value.

   Return Value: void * to a playerset array entry

4. **Function Name/Parameters:** init_Pset()

   Parameter Types: none

   Description: Creates a sector's playerset and passes back a void pointer for the sector to track the starting location of its own playerset. This void pointer actually points to a sector's hash table. The hash table is an array of 25 pointers, each of which points to a linked list of players of struct object_attributes.

   Return Value: void * to playerset

5. **Function Name/Parameters:** add_Pset_player(Pset_ptr, P_id)

   Parameter Types: void *Pset_ptr[], struct object_attributes *P_id

   Description: Adds a player to a sector's playerset. The function returns a pointer to the player which has been inserted or a NULL value if the function call was not made to a valid list or not enough space exists on the heap to hold the next player structure.

   Return Value: void * to player

6. **Function Name/Parameters:** Pset_is_empty(Pset_ptr)

   Parameter Types: void *Pset_ptr[]

   Description: Returns a true (1) or false (0) depending upon whether the specified playerset is completely empty or not, respectively.

   Return Value: int (1 or 0)

7. **Function Name/Parameters:** get_first_Pset_player(Pset_ptr)

   Parameter Types: void *Pset_ptr[]

   Description: Retrieve the FIRST player from a specified sector's player set. If the playerset is empty or a valid linked list was not passed to the function, then a NULL value is returned.

   Return Value: void * to player

8. **Function Name/Parameters:** get_next_Pset_player(Pset_ptr)

   Parameter Types: void *Pset_ptr[]

   Description: Retrieve the next player from a specified sector's player set AFTER the last one retrieved with either get_first_Pset_player or this function. If the playerset has no more players or a valid linked list was not passed to the function, then a NULL value is returned.

Return Value: void * to player

9. **Function Name/Parameters:** pop_first_Pset_player(Pset_ptr)

   Parameter Types: void *Pset_ptr[]

   Description: Return the "first" player (could be any) from a sector's playerset, and then remove it from the set entirely. If the playerset is empty, then return a flag value of NULL.

   Return Value: void * to player

10. **Function Name/Parameters:** rem_Pset_player(Pset_ptr, P_id)

    Parameter Types: void *Pset_ptr[], struct object_attributes *P_id

    Description: Removes a specified player from a sector's playerset. If the player is not deleted or if the function call was not made to a valid playerset then the function returns NULL.

    Return Value: void * to removed player

11. **Function Name/Parameters:** player_in_Pset(Pset_ptr, P_id)

    Parameter Types: void *Pset_ptr[], struct object_attributes *P_id

    Description: Checks to see if a given player already exists in a player set. If so, then it returns a true value (1); if not, then it returns a false value (0).

    Return Value: int (1 or 0)

12. **Function Name/Parameters:** initSetPlayer()

    Parameter Types: none

    Description: Initializes Soderholm's master object array with NULL values. Created for compatibility with Soderholm's version of BATTLESIM.

    Return Value: none

13. **Function Name/Parameters:** add_player(id, player)

    Parameter Types: int id, void *player

    Description: Adds a player to Soderholm's master object array. Created for compatibility with Soderholm's version of BATTLESIM.

    Return Value: none

14. **Function Name/Parameters:** get_player(id)

    Parameter Types: int id

    Description: Retrieves a specified player from Soderholm's master object array. Created for compatibility with Soderholm's version of BATTLESIM.

    Return Value: none

15. **Function Name/Parameters:** rem_player(id)

    Parameter Types: int id

Description: Returns a pointer to a player and removes it from the playerset. Created for compatibility with Soderholm's version of BATTLESIM.

Return Value: struct object_attributes * to removed player

16. **Function Name/Parameters:** setmaxLocPlayer(id)

    Parameter Types: int id

    Description: Sets the number of local players. Created for compatibility with Soderholm's version of BATTLESIM.

    Return Value: none

17. **Function Name/Parameters:** getmaxLocPlayer()

    Parameter Types: none

    Description: Retrieves the number of local players. Created for compatibility with Soderholm's version of BATTLESIM.

    Return Value: int

18. **Function Name/Parameters:** setmaxGlbPlayer(id)

    Parameter Types: int id

    Description: Sets the number of global players. Created for compatibility with Soderholm's version of BATTLESIM.

    Return Value: none

19. **Function Name/Parameters:** getmaxGlbPlayer()

    Parameter Types: none

    Description: Retrieves the number of global players. Created for compatibility with Soderholm's version of BATTLESIM.

    Return Value: int

20. **Function Name/Parameters:** show_MOA()

    Parameter Types: none

    Description: Show the contents of the master object array (MOA). Created for compatbility with Soderholm's version of BATTLESIM.

    Return Value: none

B.4 Methods for Accessing Sector Object.

This section lists all of the object-based methods developed to access, modify, and delete the fields and structures within the sector ADT. All of the methods, as well as the sector structure itself, are encapsulated within the file **sector.c**. This file is comprised of 4813 lines of code in 86 methods.

1. **Function Name/Parameters:** list_sector_neighbors(S_id)

   Parameter Types: int S_id

   Description: Displays all the neighbor sectors of a given sector. If one does not exist, then a message is displayed stating this.

   Return Value: none

2. **Function Name/Parameters:** determine_sector_neighbors(S_id)

   Parameter Types: int S_id

   Description: Computes all of the neighbors for a given sector, including those at the same level (on z axis), those above the given sector (+z axis), and those below the given sector (-z axis). If a neighbor does NOT exist whatsoever, then a dummy value of -1 is returned.

   Return Value: none

3. **Function Name/Parameters:** det_boundary_event(player1)

   Parameter Types: struct object_attributes *player1

   Description: Returns the boundary event (one will ALWAYS exist if the player exists and is moving); return NULL otherwise.

   Return Value: void *

4. **Function Name/Parameters:** time_to_intercept_bound(player, event_type, bound)

   Parameter Types: struct object_attributes *player, int *event_type, *bound

   Description: Computes the minimum time for a player to intercept either the x, y, or z-axis boundary of a given sector. The function also returns two pointers indicating which bound and what kind of boundary crossing event is associated with that minimum time.

   Return Value: double, and two int *'s

5. **Function Name/Parameters:** time_to_intercept_x_bound(player1, event_type)

   Parameter Types: struct object_attributes *player1, int *event_type

   Description: Computes time for a player's front end sensor, back end sensor, and center of mass to intersect the x-axis of a given sector. It returns the minimum of the three values, or a flag value of INFINITY if none of them intersect the x-axis. It also returns a pointer indicating which boundary crossing event is associated with this time.

   Return Value: double, and one int *

6. **Function Name/Parameters:** time_to_intercept_y_bound(player1, event_type)

   Parameter Types: struct object_attributes *player1, int *event_type

   Description: Computes time for a player's front end sensor, back end sensor, and center of mass to intersect the y-axis of a given sector. It returns the minimum of the three values, or a flag value of INFINITY if none of them intersect the y-axis.

It also returns a pointer indicating which boundary crossing event is associated with this time.

Return Value: double, and one int *

7. **Function Name/Parameters:** time_to_intercept_z_bound(player1, event_type)

   Parameter Types: struct object_attributes *player1, int *event_type

   Description: Computes time for a player's front end sensor, back end sensor, and center of mass to intersect the z-axis of a given sector. It returns the minimum of the three values, or a flag value of INFINITY if none of them intersect the z-axis. It also returns a pointer indicating which boundary crossing event is associated with this time.

   Return Value: double, and one int *

8. **Function Name/Parameters:** list_all_sectors()

   Parameter Types: none

   Description: Lists all the information contained within ALL sectors used by a given scenario, including all fields in the sector structure and all information contained in the playerset structure. USED FOR DEBUGGING.

   Return Value: none

9. **Function Name/Parameters:** list_sector(S_id)

   Parameter Types: int S_id

   Description: Lists all the information contained within a specified sector, including all the fields in the sector structure and all the information contained in the playerset structure. USED FOR DEBUGGING.

   Return Value: none

10. **Function Name/Parameters:** copy_S_player(S_id1, S_id2, obj_id1)

    Parameter Types: int S_id1, int S_id2, int obj_id1

    Description: Copies a given player in its entirety from one sector to another. This is not accomplished with pointers, but by copying each field one at a time to ensure each structure is completely independent.

    Return Value: void * to new player

11. **Function Name/Parameters:** init_sectors()

    Parameter Types: none

    Description: This function initializes the sector array, and must be called AFTER the master object array has been completely initialized. When finished, the sector array will contain all info required to begin a partitioned battlefield simulation. Steps accomplished in this function include:

    - generating playersets

- reading in sector information from the scenario file, and set all fields in the sector structure
- determining what players to copy into each sector's playerset, and doing it

Return Value: none

12. **Function Name/Parameters:** create_S_Pset(S_id)

    Parameter Types: int S_id

    Description: Creates a sector's playerset by generating linked lists for the hash table.

    Return Value: none

13. **Function Name/Parameters:** get_S_player(S_id)

    Parameter Types: int S_id

    Description: Searches through the hash table of players until the correct player structure is found. If the player does not exist in the provided playerset, then a value of NULL is returned.

    Return Value: void * to desired player structure

14. **Function Name/Parameters:** player_copy_in_S(S_id, P_id)

    Parameter Types: int S_id, int P_id

    Description: Checks to see if a copy of a player exists in the sector's playerset. If it does, then return true (1); else return false (0).

    Return Value: int (1 or 0)

15. **Function Name/Parameters:** in_sector_bounds(S_id, P_id)

    Parameter Types: int S_id, struct object_attributes *P_id

    Description: Checks to see if a given player is physically located within a given sector's boundaries. If so, then return true (1); else return false (0).

    Return Value: int (1 or 0)

16. **Function Name/Parameters:** init_S_bounds(S_id)

    Parameter Types: int S_id

    Description: Initializes all of the sector boundaries based upon the contents of a sector definition file.

    Return Value: none

17. **Function Name/Parameters:** getSleft_neighbor(S_id)

    Parameter Types: int S_id

    Description: Retrieves the sector_id of a given sector's left neighbor.

    Return Value: int

18. **Function Name/Parameters:** setSleft_neighbor(S_id, sector_id)

    Parameter Types: int S_id, int sector_id

    Description: Sets the sector_id of a given sector's left neighbor.

    Return Value: none

19. **Function Name/Parameters:** getSright_neighbor(S_id)

    Parameter Types: int S_id

    Description: Retrieves the sector_id of a given sector's right neighbor.

    Return Value: int

20. **Function Name/Parameters:** setSright_neighbor(S_id, sector_id)

    Parameter Types: int S_id, int sector_id

    Description: Sets the sector_id of a given sector's right neighbor.

    Return Value: none

21. **Function Name/Parameters:** getStop_neighbor(S_id)

    Parameter Types: int S_id

    Description: Retrieves the sector_id of a given sector's top neighbor.

    Return Value: int

22. **Function Name/Parameters:** setStop_neighbor(S_id, sector_id)

    Parameter Types: int S_id, int sector_id

    Description: Sets the sector_id of a given sector's top neighbor.

    Return Value: none

23. **Function Name/Parameters:** getSbottom_neighbor(S_id)

    Parameter Types: int S_id

    Description: Retrieves the sector_id of a given sector's bottom neighbor.

    Return Value: int

24. **Function Name/Parameters:** setSbottom_neighbor(S_id, sector_id)

    Parameter Types: int S_id, int sector_id

    Description: Sets the sector_id of a given sector's bottom neighbor.

    Return Value: none

25. **Function Name/Parameters:** getStop_left_neighbor(S_id)

    Parameter Types: int S_id

    Description: Retrieves the sector_id of a given sector's top_left neighbor.

    Return Value: int

26. **Function Name/Parameters:** setStop_left_neighbor(S_id, sector_id)

    Parameter Types: int S_id, int sector_id

    Description: Sets the sector_id of a given sector's top_left neighbor.

    Return Value: none

27. **Function Name/Parameters:** getStop_right_neighbor(S_id)

    Parameter Types: int S_id

    Description: Retrieves the sector_id of a given sector's top_right neighbor.

    Return Value: int

28. **Function Name/Parameters:** setStop_right_neighbor(S_id, sector_id)

    Parameter Types: int S_id, int sector_id

    Description: Sets the sector_id of a given sector's top_right neighbor.

    Return Value: none

29. **Function Name/Parameters:** getSbottom_left_neighbor(S_id)

    Parameter Types: int S_id

    Description: Retrieves the sector_id of a given sector's bottom_left neighbor.

    Return Value: int

30. **Function Name/Parameters:** setSbottom_left_neighbor(S_id, sector_id)

    Parameter Types: int S_id, int sector_id

    Description: Sets the sector_id of a given sector's bottom_left neighbor.

    Return Value: none

31. **Function Name/Parameters:** getSbottom_right_neighbor(S_id)

    Parameter Types: int S_id

    Description: Retrieves the sector_id of a given sector's bottom_right neighbor.

    Return Value: int

32. **Function Name/Parameters:** setSbottom_right_neighbor(S_id, sector_id)

    Parameter Types: int S_id, int sector_id

    Description: Sets the sector_id of a given sector's bottom_right neighbor.

    Return Value: none

33. **Function Name/Parameters:** getSdirectly_over_neighbor(S_id)

    Parameter Types: int S_id

    Description: Retrieves the sector_id of a given sector's neighbor that is directly over it in the z axis.

    Return Value: int

34. **Function Name/Parameters:** setSdirectly_over_neighbor(S_id, sector_id)

    Parameter Types: int S_id, int sector_id

    Description: Sets the sector_id of a given sector's neighbor that is directly over it in the z axis.

    Return Value: none

35. **Function Name/Parameters:** getSo_left_neighbor(S_id)

    Parameter Types: int S_id

    Description: Retrieves the sector_id of a given sector's left neighbor over it (+z axis).

    Return Value: int

36. **Function Name/Parameters:** setSo_left_neighbor(S_id, sector_id)

    Parameter Types: int S_id, int sector_id

    Description: Sets the sector_id of a given sector's left neighbor over it (+z axis).

    Return Value: none

37. **Function Name/Parameters:** getSo_right_neighbor(S_id)

    Parameter Types: int S_id

    Description: Retrieves the sector_id of a given sector's right neighbor over it (+z axis).

    Return Value: int

38. **Function Name/Parameters:** setSo_right_neighbor(S_id, sector_id)

    Parameter Types: int S_id, int sector_id

    Description: Sets the sector_id of a given sector's right neighbor over it (+z axis).

    Return Value: none

39. **Function Name/Parameters:** getSo_top_neighbor(S_id)

    Parameter Types: int S_id

    Description: Retrieves the sector_id of a given sector's top neighbor over it (+z axis).

    Return Value: int

40. **Function Name/Parameters:** setSo_top_neighbor(S_id, sector_id)

    Parameter Types: int S_id, int sector_id

    Description: Sets the sector_id of a given sector's top neighbor over it (+z axis).

    Return Value: none

41. **Function Name/Parameters:** getSo_bottom_neighbor(S_id)

    Parameter Types: int S_id

    Description: Retrieves the sector_id of a given sector's bottom neighbor over it (+z axis).

    Return Value: int

42. **Function Name/Parameters:** setSo_bottom_neighbor(S_id, sector_id)

    Parameter Types: int S_id, int sector_id

    Description: Sets the sector_id of a given sector's bottom neighbor over it (+z axis).

    Return Value: none

43. **Function Name/Parameters:** getSo_top_left_neighbor(S_id)

    Parameter Types: int S_id

    Description: Retrieves the sector_id of a given sector's top_left neighbor over it (+z axis).

    Return Value: int

44. **Function Name/Parameters:** setSo_top_left_neighbor(S_id, sector_id)

    Parameter Types: int S_id, int sector_id

    Description: Sets the sector_id of a given sector's top_left neighbor over it (+z axis).

    Return Value: none

45. **Function Name/Parameters:** getSo_top_right_neighbor(S_id)

    Parameter Types: int S_id

    Description: Retrieves the sector_id of a given sector's top_right neighbor over it (+z axis).

    Return Value: int

46. **Function Name/Parameters:** setSo_top_right_neighbor(S_id, sector_id)

    Parameter Types: int S_id, int sector_id

    Description: Sets the sector_id of a given sector's top_right neighbor over it (+z axis).

    Return Value: none

47. **Function Name/Parameters:** getSo_bottom_left_neighbor(S_id)

    Parameter Types: int S_id

    Description: Retrieves the sector_id of a given sector's bottom_left neighbor over it (+z axis).

    Return Value: int

48. **Function Name/Parameters:** setSo_bottom_left_neighbor(S_id, sector_id)

    Parameter Types: int S_id, int sector_id

    Description: Sets the sector_id of a given sector's bottom_left neighbor over it (+z axis).

    Return Value: none

49. **Function Name/Parameters:** getSo_bottom_right_neighbor(S_id)

    Parameter Types: int S_id

    Description: Retrieves the sector_id of a given sector's bottom_right neighbor over it (+z axis).

    Return Value: int

50. **Function Name/Parameters:** setSo_bottom_right_neighbor(S_id, sector_id)

    Parameter Types: int S_id, int sector_id

    Description: Sets the sector_id of a given sector's bottom_right neighbor over it (+z axis).

    Return Value: none

51. **Function Name/Parameters:** getSdirectly_under_neighbor(S_id)

    Parameter Types: int S_id

    Description: Retrieves the sector_id of a given sector's neighbor that is directly under it in the z axis.

    Return Value: int

52. **Function Name/Parameters:** setSdirectly_under_neighbor(S_id, sector_id)

    Parameter Types: int S_id, int sector_id

    Description: Sets the sector_id of a given sector's neighbor that is directly under it in the z axis.

    Return Value: none

53. **Function Name/Parameters:** getSu_left_neighbor(S_id)

    Parameter Types: int S_id

    Description: Retrieves the sector_id of a given sector's left neighbor under it (-z axis).

    Return Value: int

54. **Function Name/Parameters:** setSu_left_neighbor(S_id, sector_id)

    Parameter Types: int S_id, int sector_id

    Description: Sets the sector_id of a given sector's left neighbor under it (-z axis).

    Return Value: none

55. **Function Name/Parameters:** getSu_right_neighbor(S_id)

    Parameter Types: int S_id

    Description: Retrieves the sector_id of a given sector's right neighbor under it (-z axis).

    Return Value: int

56. **Function Name/Parameters:** setSu_right_neighbor(S_id, sector_id)

    Parameter Types: int S_id, int sector_id

    Description: Sets the sector_id of a given sector's right neighbor under it (-z axis).

    Return Value: none

57. **Function Name/Parameters:** getSu_top_neighbor(S_id)

    Parameter Types: int S_id

    Description: Retrieves the sector_id of a given sector's top neighbor under it (-z axis).

    Return Value: int

58. **Function Name/Parameters:** setSu_top_neighbor(S_id, sector_id)

    Parameter Types: int S_id, int sector_id

    Description: Sets the sector_id of a given sector's top neighbor under it (-z axis).

    Return Value: none

59. **Function Name/Parameters:** getSu_bottom_neighbor(S_id)

    Parameter Types: int S_id

    Description: Retrieves the sector_id of a given sector's bottom neighbor under it (-z axis).

    Return Value: int

60. **Function Name/Parameters:** setSu_bottom_neighbor(S_id, sector_id)

    Parameter Types: int S_id, int sector_id

    Description: Sets the sector_id of a given sector's bottom neighbor under it (-z axis).

    Return Value: none

61. **Function Name/Parameters:** getSu_top_left_neighbor(S_id)

    Parameter Types: int S_id

    Description: Retrieves the sector_id of a given sector's top_left neighbor under it (-z axis).

    Return Value: int

62. **Function Name/Parameters:** setSu_top_left_neighbor(S_id, sector_id)

    Parameter Types: int S_id, int sector_id

    Description: Sets the sector_id of a given sector's top_left neighbor under it (-z axis).

    Return Value: none

63. **Function Name/Parameters:** getSu_top_right_neighbor(S_id)

    Parameter Types: int S_id

    Description: Retrieves the sector_id of a given sector's top_right neighbor under it (-z axis).

    Return Value: int

64. **Function Name/Parameters:** setSu_top_right_neighbor(S_id, sector_id)

    Parameter Types: int S_id, int sector_id

    Description: Sets the sector_id of a given sector's top_right neighbor under it (-z axis).

    Return Value: none

65. **Function Name/Parameters:** getSu_bottom_left_neighbor(S_id)

    Parameter Types: int S_id

    Description: Retrieves the sector_id of a given sector's bottom_left neighbor under it (-z axis).

    Return Value: int

66. **Function Name/Parameters:** setSu_bottom_left_neighbor(S_id, sector_id)

    Parameter Types: int S_id, int sector_id

    Description: Sets the sector_id of a given sector's bottom_left neighbor under it (-z axis).

    Return Value: none

67. **Function Name/Parameters:** getSu_bottom_right_neighbor(S_id)

    Parameter Types: int S_id

    Description: Retrieves the sector_id of a given sector's bottom_right neighbor under it (-z axis).

    Return Value: int

68. **Function Name/Parameters:** setSu_bottom_right_neighbor(S_id, sector_id)

    Parameter Types: int S_id, int sector_id

    Description: Sets the sector_id of a given sector's bottom_right neighbor under it (-z axis).

    Return Value: none

69. **Function Name/Parameters:** getSid(S_id)

    Parameter Types: int S_id

    Description: Retrieve the sector's sector identifier.

    Return Value: int

70. **Function Name/Parameters:** setSid(S_id, nbr)

    Parameter Types: int S_id, int nbr

    Description: Sets the sector's sector identifier.

    Return Value: none

71. **Function Name/Parameters:** getSminval_x(S_id)

    Parameter Types: int S_id

    Description: Retrieves the minimum x_value for sector.

    Return Value: double

72. **Function Name/Parameters:** setSminval_x(S_id, nbr)

    Parameter Types: int S_id, int nbr

    Description: Sets the minimum x_value for this sector.

    Return Value: none

73. **Function Name/Parameters:** getSminval_y(S_id)

    Parameter Types: int S_id

    Description: Retrieves the minimum y_value for sector.

    Return Value: double

74. **Function Name/Parameters:** setSminval_y(S_id, nbr)

    Parameter Types: int S_id, int nbr

    Description: Sets the minimum y_value for this sector.

    Return Value: none

75. **Function Name/Parameters:** getSminval_z(S_id)

    Parameter Types: int S_id

    Description: Retrieves the minimum z_value for sector.

    Return Value: double

76. **Function Name/Parameters:** setSminval_z(S_id, nbr)

    Parameter Types: int S_id, int nbr

    Description: Sets the minimum z_value for this sector.

    Return Value: none

77. **Function Name/Parameters:** getSmaxval_x(S_id)

    Parameter Types: int S_id

    Description: Retrieves the maximum x_value for sector.

    Return Value: double

78. **Function Name/Parameters:** setSmaxval_x(S_id, nbr)

    Parameter Types: int S_id, int nbr

    Description: Sets the maximum x_value for this sector.

    Return Value: none

79. **Function Name/Parameters:** getSmaxval_y(S_id)

    Parameter Types: int S_id

    Description: Retrieves the maximum y_value for sector.

    Return Value: double

80. **Function Name/Parameters:** setSmaxval_y(S_id, nbr)

    Parameter Types: int S_id, int nbr

    Description: Sets the maximum y_value for this sector.

    Return Value: none

81. **Function Name/Parameters:** getSmaxval_z(S_id)

    Parameter Types: int S_id

    Description: Retrieves the maximum z_value for sector.

    Return Value: double

82. **Function Name/Parameters:** setSmaxval_z(S_id, nbr)

    Parameter Types: int S_id, int nbr

    Description: Sets the maximum z_value for this sector.

    Return Value: none

83. **Function Name/Parameters:** getPset(S_id)

    Parameter Types: int S_id

    Description: Retrieves the pointer to the playerset for a given sector.

    Return Value: void *

84. **Function Name/Parameters:** setPset(S_id)

    Parameter Types: int S_id

    Description: Sets the pointer to the playerset for a given sector.

    Return Value: none

85. **Function Name/Parameters:** getSnet(S_id)
    Parameter Types: int S_id
    Description: Retrieves the next_event_time for a given sector.
    Return Value: double
86. **Function Name/Parameters:** setSnet(S_id)
    Parameter Types: int S_id
    Description: Sets the next_event_time for a given sector.
    Return Value: none

B.5 Example of Using Major BATTLESIM Methods.

The implementation of the player, playerset, and sector objects just shown provides the "hierarchical approach" necessary to ensure that software engineering principles such as those described in the introduction are met. For example, let's say BATTLESIM needs to retrieve the object_type field of player 28 which is located in sector 8. The following sequence of events — which have been simplified for the sake of clarity — occurs:

- BATTLESIM calls **`get_S_player`**, a sector method
- **`get_S_player`** calls **`Pset_is_empty`** — a playerset method — to determine whether the playerset is empty or not. If it isn't empty, it calls **`get_hash_entry`** — another playerset method — to try and access the sector's playerset.
- **`get_hash_entry`** calls **`ll_get_data`** — a linked list method — to retrieve the pointer to the actual player structure in the playerset.
- **`ll_get_data`** calls the player method **`getPobj_id`** to retrieve the desired information.

Appendix C. *Compendium of Resolved Errors and Limitations*

C.1 Introduction.

This appendix summarizes the deficiencies discovered in the last version of BATTLESIM — whether they were known before this phase of research began or not — and what was done to resolve them. The entries listed here are for errors which were known to cause execution errors, as well as perceived limitations in software design or implementation which could be handled in a more effective manner.

Each entry is only intended to serve as a quick reference of one of the differences that exist between Capt Soderholm's version of BATTLESIM and the current version, and briefly elaborate on what change was implemented to resolve the problem — it is *not* intended to discuss all the possible options or why the particular one used was indeed selected. For further elaboration on why a particular technique was chosen over others, the reader is referred to the previous chapters, especially Chapters 4 and 6.

C.2 Errors/Limitations.

1. **Error/Limitation:** Could only use one scenario input file for a given scenario, no matter how many LPs were being used in a given scenario.

   Description of Resolution: Allow the capability for the user to specify whether he wants to use just one scenario file for all LPs, or one that was designed to be used by only one LP in the scenario.

2. **Error/Limitation:** The code for Version 2 of the TCHSIM simulation driver was closely entangled into the other two major software components in BATTLESIM: the application code and the SPECTRUM simulation driver. This prevented upgrading to the latest version of TCHSIM, which contained additional enhancements needed in the current version of BATTLESIM, as well as hindering the ability for future upgrades as well.

Description of Resolution: Disentangled Version 2 of TCHSIM from BATTLESIM in a modular manner which not only supported the upgrade to Version 3, but that also ensures that future upgrades can be made more easily.

3. **Error/Limitation:** A scenario could only be run in parallel on one or more nodes of the Intel Hypercube.

   Description of Resolution: Redesign BATTLESIM to utilize a hierarchical approach in which hardware-specific functions are kept isolated at the lowest levels of BATTLESIM in hardware-specific files. Current version runs sequentially on the Sun Sparcstation, sequentially on the host of the Intel Hypercube, and in parallel using one or more nodes of the Hypercube.

4. **Error/Limitation:** All player state information was replicated on all nodes in a scenario, incurring unnecessary message-passing overhead.

   Description of Resolution: Eliminate all unnecessary player state information from each node, so that it only has knowledge of those players it requires.

5. **Error/Limitation:** Each LP in a scenario had visibility to the entire battlefield, which increased the size of the state space each player had to perform sensor calculations against.

   Description of Resolution: Partition the battlefield into sectors, and assign responsibility for each sector to one LP which controls it for the life of the scenario. Each player only has to perform sensor calculations against the reduced search space of sectors the player has visibility into.

6. **Error/Limitation:** Each player had to be sequentially numbered with an integer value, beginning with one. No "skipping" of values was allowed, forcing BATTLESIM to remember the last value used.

   Description of Resolution: Allow each player to be uniqely identified with any integer value, as long as it does not exceed the integer capacity of the machine on which it is running and has not already been used.

7. **Error/Limitation:** An integer array of size 1024 was used to contain all players in the scenario, limiting the total number of players allowed in each simulation run to 1024.

   Description of Resolution: Use a linked list to hold the players, allowing as many players in each scenario as the simulation platform's memory can support.

8. **Error/Limitation:** Lacked methods necessary to support creation, deletion and modification of the abstract data types and other structures in BATTLESIM. This forced the user to access low-level fields directly, thereby severely limiting the ability to transparently change the structures containing these fields.

   Description of Resolution: Ensure that all major abstract data types in the present version of BATTLESIM — the player, playerset, and sector objects — all have methods defined to access and modify them.

9. **Error/Limitation:** Each record in the single input scenario file was read by BATTLESIM into a fixed-length buffer and parsed, with each field in the record possessing a variable-length; this buffer limited the number of bytes each record in the scenario file could hold.

   Description of Resolution: Read each field of each record into a record of variable length itself. This would allow each field in the input to as long as desired, which was the original intention since it was defined to be variable-length originally.

10. **Error/Limitation:** The DeRouchey output file binds each object to its associated object type *after* all types have been identified, all at the end of the scenario run. This forces it to try and bind each and every object type.

    Description of Resolution: Identify all types in the scenario input file(s), and bind only the objects that require it during the simulation.

11. **Error/Limitation:** Route points for each player must be specified in reverse order in the scenario files because the route points are stored in a last-in-first-out (LIFO) queue. This is a user inconvenience.

    Description of Resolution: Update the BATTLESIM function which reads all the scenario files to reverse the route points itself after they have been read. This makes

the structure used to hold route points transparent, and allows users to input points from the first one to the last one as they would naturally.

12. **Error/Limitation:** Players can only collide with each other when their center of mass' precisely overlap. This is unrealistic, especially for larger players.

    Description of Resolution: Add a radius attribute to players. This not only allows players to collide with each other when their center of mass' don't overlap, but it realistically makes this occur earlier for larger aircraft.

13. **Error/Limitation:** Current equations for player motion do not support player acceleration, unduly limiting future enhancements to player fidelity.

    Description of Resolution: Update equations for player motion to contain support for player acceleration in case it is used.

14. **Error/Limitation:** Inability to provide information to BATTLESIM at run-time in a way which is amenable to the addition of new parameters without requiring recompilation of the application.

    Description of Resolution: Add command-line arguments which augment the information provided by the scenario files, and don't require the application to be recompiled.

15. **Error/Limitation:** Each player must be moving on the battlefield in order to sense other players. Otherwise, it has no sensor capability whatsoever.

    Description of Resolution: Allow player to have sensing capabilities whether they are moving or not. If the object has a non-zero velocity with no routepoints, a problem still exists. However, if the player has no velocity with no routepoints, then no problem exists.

16. **Error/Limitation:** No methods exist in the linked list package to retrieve data without popping it off the linked list entirely.

    Description of Resolution: Create methods which allow the user to traverse a linked list and retrieve either an entry's data, or its pointer location, thereby alleviating the need to pop each data item off.

Appendix D. *BATTLESIM Configuration Guide*

D.1 Software Files.

The software files supporting the current version of BATTLESIM, listed in alphabetical order for quick reference, include the following:

- application.h
- battle.c and battle.h
- cube2.c and cube2.h
- dll.c
- filters.c
- globals.h
- icon.c
- interfaceB.c
- ll.c and ll.h
- lp_man.c
- message.c and message.h
- myfilters.c
- player.c and player.h
- playerset.c and playerset.h
- proc1.arcs
- protocol.c
- sector.c and sector.h
- sensor.c
- sim_func.c and sim_func.h
- sim_read.c and sim_read.h

- sim_stru.h
- tchmap.c
- terrain.c
- use_visit.c
- scenario files
- map files
- Makefile

The functional description of each of these software files is provided in the next section.

D.2 Functional Description.

application.h - This file contains conditional compilation 'defines' which pass on information to BATTLESIM. Fields of interest include:

- NUM_PROCS - the number of LPs, not nodes, which a run will use. This value *must* match the number of LPs specified to the 'host' program when executing a scenario, or else the run will abnormally terminate.
- INPUT_ARCS - the file specifying the SPECTRUM communication arcs between BATTLESIM LPs. A file must be specified, even though it is not actively used, because SPECTRUM requires it.
- MAXTIME - specifies a run's maximum allowable execution time in milliseconds

battle.c -`Battle.c`, formerly known as `rizsim.c`, is one of two main BATTLESIM application files. It contains application-specific functions which are independent of the hardware platform and SPECTRUM, such as:

- BATTLESIM initialization
- scheduling initial player events
- all the event-handling routines

- starting and stopping the user's screen output

battle.h - Contains numeric identifiers for all the event types in BATTLESIM; including the three new boundary-crossing events. It also holds two important conditional compilation 'defines', SCREEN_DISPLAY and GRAPHIC_FILE. If SCREEN_DISPLAY is commented out and BATTLESIM is recompiled, then no output is sent to the user's screen. Likewise, if GRAPHIC_FILE is commented out and BATTLESIM is recompiled, then no output is sent to the graphics file `display.out`. These are normally left intact to generate both screen output and the graphics file.

cube2.c - This file contains all the Intel Hypercube-specific functions necessary to support BATTLESIM. It is the lowest level of communications between LPs in the 'hierarchical approach' implemented by BATTLESIM. Soderholm's message-passing protocol — comprised of the functions node_send_one_message and node_get_one_message — is kept in here. His approach bypassed SPECTRUM to make direct calls to the Hypercube.

cube2.h - This file defines Intel message types, and contains a table definition which maps the LP identifiers to the Hypercube node and process numbers. This tells each node process how to send a message to any other LP.

dll.c - This package is very similar to Rizza's linked list package, except this package was designed by Soderholm to support doubly linked lists, i.e. linked lists with both head and tail pointers so it can be traversed in either direction.

filters.c - another SPECTRUM file supposedly describing the various time synchronization protocols used by BATTLESIM. There currently are none. The only SPECTRUM file that appears to reference this file is the Makefile itself, but it is kept because the 'user interface' utility builds this file for any provided filter set (13). The actual filter file for BATTLESIM is `myfilters.c`.

globals.h - SPECTRUM supports only one type of message for transferring information from one LP to another. This event-type structure definition is stored within this file, and contains at least the following fields:

1. a message time-stamp

2. the event type
3. line number over which it is sent
4. identifiers of the source and destination LPs (13)

While additional state information fields specific to a given application can be added as long as no existing fields are changed or deleted, BATTLESIM currently adds no fields of its own. However, since this definition is used throughout SPECTRUM, all SPECTRUM software files should be recompiled if this file is modified in any manner.

icon.c - An object-based icon management package developed by Mr. Rick Norris. It utilizes a linked list to hld all the icons used to represent players in BATTLESIM, tracking both their number and name.

interfaceB.c - The BATTLESIM version of the SPECTRUM file `interface1.c`. This file acts as the "link" between the BATTLESIM application and the lower-level SPECTRUM and Hypercube-specific functions, by containing all calls to the encapsulated lower-level structures (15:12).

ll.c and ll.h - A linked list package designed to support LIFO, FIFO, and priority queues containing any kind of data structure desired. This package is used extensively to build and maintain several BATTLESIM structures, including the six linked lists contained within the 'player' definition and the buckets in the playerset's hash table. The file `ll.h` contains the declarations for the methods in `ll.c`.

lp_man.c - Contains the code implementing the SPECTRUM logical process manager, which is both application and machine-independent. The LP manager maintains the input queue of messages from other LPs in simulation time-stamp order. BATTLESIM only uses the initialization functions.

message.c and message.h - An object-based package designed to pack, list, and unpack player messages sent between Hypercube nodes. Each message contain all the fields and structures contained within a player, allowing players to be either updated or created on the receiving node. The file `message.h` contains the message package's method declarations.

myfilters.c - the SPECTRUM file which actually holds any 'filters' used. There are currently no filters used by BATTLESIM.

player.c and player.h - The files containing the object-based definition of a BATTLESIM player. All the methods necessary to retrieve, modify, and delete all the fields in a player without knowledge of the player's underlying structure are kept here. See Appendix B for a listing and description of those methods.

procl.arcs - LPs in SPECTRUM communicate via unidirectional lines known as *arcs*. This file tells SPECTRUM which LPs communicate with each other, i.e. it describes BATTLESIM's communications 'network'. Even though this file contains no entries which are actively used, SPECTRUM still requires it to exist.

playerset.c and playerset.h - The files which contain the object-based definition of a BATTLESIM playerset, presently implemented as an open hash table with buckets composed of linked lists. All the methods necessary to retrieve, modify, and delete the playerset without knowledge of the playerset's underlying structure are stored here. The user should refer to Appendix B for a complete listing and description of these methods.

protocol.c - This file implements the conservative time synchronization algorithm used by BATTLESIM. The components previously used to support the optimistic time synchronization protocol have been removed.

sector.c and sector.h - The two files which hold the object-based definition of a BATTLESIM sector. The sector "container" object, a 64-entry array, is also here. All the methods used to retrieve, modify, and delete the fields in the sector via the sector array without knowledge of the underlying structures are kept here. The user can see a complete listing of all sector methods, along with their associated descriptions, in Appendix B.

sensor.c - Holds all the methods associated with determining a specified player's next event. Each kind of potential next event is computed and returned for comparison with the best event computed thus far — the one with the lowest next event time is instantiated by adding it to the appropriate next event queue. Supporting methods for solving quadratic motion equations and determining player intercept times are included as well.

sim_func.c and sim_func.h - One of two main BATTLESIM application files. It contains application-specific functions which are independent of the Hypercube and SPECTRUM.

sim_read.c and sim_read.h - These files contain the functions which read the data from an LP's scenario file, and store it in the appropriate location. For route points, it reverses those read so they are stored in reverse order as required.

sim_stru.h - This file contains the structural definition of a BATTLESIM player.

tchmap.c - A TCHSIM file which contains an object-based implementation of an object-to-LP map. Each map consists of a set of object instances to logical processes. This is used by BATTLESIM to track sector-to-LP assignments.

terrain.c - An object-based implementation of a terrain file, used to let BATTLESIM know certain required battlefield characteristics like minimum and maximum battlefield coordinates.

use_visit.c - This file acts as an interface to VISIT, the visual graphics driver program designed by DeRouchey to display graphics output files created by BATTLESIM. Specifically, it contains functions which generate records in the graphics file to start VISIT, stop VISIT, and change the visual status of players.

scenario files - These files, whose names end with a .in4 extension, are used to convey battlefield and player state information associated with a given battlefield scenario. Each LP must read a scenario file during initialization. That scenario file may be designed for use by only one LP, or may in fact be shared by multiple LPs.

map files - Each BATTLESIM scenario requires that a map file, whose name begins with the .map extension, be provided describing the sector-to-LP assignments. Since this assignment is static for the duration of the scenario, any player entering a sector owned by a given LP is controlled by that LP while in the sector.

Makefile - This file provides an automated means of compiling and linking all the software files necessary to execute BATTLESIM. The current Makefile is contained in Appendix E.

Appendix E. *Complete BATTLESIM Execution Example*

E.1 Introduction.

This appendix contains a complete example of how to run a scenario on the Intel Hypercube using BATTLESIM. The following steps must be performed to run a BATTLESIM scenario.

- Generate the necessary scenario and map files.
- Compile the code associated with BATTLESIM to generate an executable file.
- Invoke a program to load the Hypercube with SPECTRUM, BATTLESIM, and the appropriate command-line arguments.

These steps are discussed in section E.2; the output sent to the user's terminal is shown in section E.3. The log files generated by the scenario are listed in section E.4. Lastly, the contents of the graphics output file `display.out` is shown in section E.5.

The scenario used for this example is Benchmark Scenario 13; the scenario and map files associated with this scenario are documented in Appendix A.

E.2 How to Invoke BATTLESIM.

To begin execution of Benchmark Scenario 13, the user must first log onto the Intel Hypercube. Then the scenario and map files associated with this particular scenario must be generated if they have not already, using any text editor following the prescribed file formats (see Chapter 6 for file format descriptions). The files in this scenario, which are contained in Appendix A, consist of 8 scenario files and 1 map file. Note that since there is one scenario file for LP, the '-m' option will have to be used when invoking BATTLESIM. Since LP0 is the only LP containing players at startup time, the file **bench130.in4** is the only file with player definitions. The map file **battlesim13.map** maps all 8 sectors to LP0, meaning that the other 7 LPs in the simulation (LP1-LP7) will not have control over any sectors, and therefore any players whatsoever.

Next, the files containing BATTLESIM's application code must be compiled and linked. The Makefile created to compile and link the version of BATTLESIM which runs on the Hypercube's nodes is shown next.

```
SpecOBJS = cube2.o lp_man.o filters.o myfilters.o
TCHOBJS = simdrive.o tchmap.o
RizOBJS  = dll.o ll.o
BATTLEOBJS  = sim_func.o sim_read.o battle.o terrain.o icon.o use_visit.o
              player.o playerset.o sector.o sensor.o message.o
# BATTLEOBJS  = events.o sim_func.o sim_read.o battle.o terrain.o icon.o
                use_visit.o
SODEROBJS  = interfaceB.o
RizLIB = -lm

NEWPATH = /usr/simulate/battlesim/new
BATTLEPATH = /usr/simulate/battlesim/source
BATTLEINCL = /usr/simulate/battlesim/source/include
SODPATH = /usr/simulate/rizsim/soderholm
RIZPATH = /usr/simulate/rizsim
AFITPATH = /usr/simulate/spectrum/afit
AFITINCL = /usr/simulate/spectrum/afit/include
UVAPATH = /usr/simulate/spectrum/uva
UVAOLDPATH = /usr/simulate/spectrum/uva/old
TCHPATH = /usr/simulate/tchsim
TCHINCL = /usr/simulate/tchsim/include
BERGPATH = /usr2/eng/kbergman/batlsim

all: host battlesim

host: ${AFITPATH}/host2.c
cc -o host -g -I${AFITINCL} -I${AFITPATH} ${AFITPATH}/host2.c -host

battlesim: $(RizOBJS) ${TCHPATH}/simdrive.o ${BERGPATH}/tchmap.o ${TCHPATH}/clock.o ${TCHPATH}/neqA.o
          $(SpecOBJS) $(BATTLEOBJS) ${SODEROBJS}
cc -o battlesim $(RizOBJS) ${TCHPATH}/simdrive.o ${BERGPATH}/tchmap.o ${TCHPATH}/clock.o
              $(TCHPATH)/neqA.o ${TCHPATH}/event.o  $(SpecOBJS) $(RizLIB) $(BATTLEOBJS) ${SODEROBJS} -node

# BATTLEOBJS:
sensor.o: ${BERGPATH}/sensor.c ${BERGPATH}/sim_read.h ${BERGPATH}/sector.h
cc -c ${BERGPATH}/sensor.c

sim_func.o: ${BERGPATH}/sim_stru.h ${BERGPATH}/ll.h ${BERGPATH}/sim_func.c ${BATTLEINCL}/battle.h
            ${BERGPATH}/application.h ${BATTLEINCL}/globals.h
# cc -c -I${SODPATH} -I${BATTLEINCL} ${BATTLEPATH}/sim_func.c
cc -c -I${BERGPATH} -I${BATTLEINCL} ${BERGPATH}/sim_func.c
sim_read.o: ${BERGPATH}/sim_stru.h ${BATTLEINCL}/route_pt.h ${BERGPATH}/sim_read.h ${BERGPATH}/sim_read.c
            ${BERGPATH}/application.h ${BATTLEINCL}/globals.h
# cc -c -I${BATTLEPATH} -I${SODPATH} ${BATTLEPATH}/sim_read.c
cc -c -I${BATTLEINCL} -I${SODPATH} ${BERGPATH}/sim_read.c
battle.o: ${BERGPATH}/sim_stru.h ${BATTLEINCL}/battle.h ${BERGPATH}/battle.c
# cc -c -I${SODPATH} ${BATTLEPATH}/battle.c
cc -c -I${BATTLEINCL} ${BERGPATH}/battle.c
terrain.o: ${BATTLEPATH}/terrain.c
cc -c ${BATTLEPATH}/terrain.c
icon.o: ${BATTLEPATH}/icon.c
cc -c ${BATTLEPATH}/icon.c
use_visit.o: ${BERGPATH}/sim_stru.h ${BATTLEINCL}/battle.h ${BATTLEPATH}/use_visit.c
```

```
cc -c -I${BERGPATH} -I${BATTLEINCL} ${BATTLEPATH}/use_visit.c
sector.o: ${BERGPATH}/sim_stru.h ${BATTLEINCL}/route_pt.h ${BERGPATH}/sector.h ${BERGPATH}/sector.c
cc -c -I${BATTLEINCL} ${BERGPATH}/sector.c
playerset.o: ${BATTLEINCL}/route_pt.h ${BERGPATH}/sim_stru.h ${BERGPATH}/playerset.h ${BERGPATH}/playerset.c
# cc -c ${BATTLEPATH}/playerset.c
cc -c -I${BATTLEINCL} -I${SODPATH} ${BERGPATH}/playerset.c
player.o: ${BERGPATH}/sim_stru.h ${BATTLEINCL}/route_pt.h ${BERGPATH}/player.h ${BERGPATH}/player.c
cc -c -I${BERGPATH} -I${BATTLEINCL} -I${SODPATH} ${BERGPATH}/player.c

ll.o: ${BERGPATH}/ll.h ${BATTLEINCL}/route_pt.h ${BERGPATH}/ll.c
# cc -c -I${SODPATH} ${SODPATH}/ll.c
cc -c -I${BATTLEINCL} -I${BATTLEPATH} -I${SODPATH} ${BERGPATH}/ll.c

tchmap.o: ${BERGPATH}/tchmap.c
# cc -c ${TCHPATH}/tchmap.c
cc -c ${BERGPATH}/tchmap.c

message.o: ${BERGPATH}/sim_stru.h ${BATTLEINCL}/route_pt.h ${BERGPATH}/message.c ${BERGPATH}/message.h
        ${BERGPATH}/player.h ${BERGPATH}/ll.h
# cc -c ${BERGPATH}/message.c
cc -c -I${BATTLEINCL} ${BERGPATH}/message.c


cube2.o: ${AFITPATH}/cube2.c ${BERGPATH}/application.h ${BATTLEINCL}/globals.h ${AFITINCL}/cube2.h
cc -c -I${BERGPATH} -I${BATTLEINCL} -I${AFITINCL} ${AFITPATH}/cube2.c
lp_man.o:${UVAPATH}/lp_man.c ${BERGPATH}/application.h ${BATTLEINCL}/globals.h
cc -c -I${BERGPATH} -I${BATTLEINCL} ${UVAOLDPATH}/lp_man.c
myfilters.o: ${BATTLEPATH}/myfilters.c ${BERGPATH}/application.h ${BATTLEINCL}/globals.h
cc -c -I${BERGPATH} -I${BATTLEINCL} ${BATTLEPATH}/myfilters.c
filters.o: ${BATTLEPATH}/filters.c ${BERGPATH}/application.h ${BATTLEINCL}/globals.h
cc -c -I${BERGPATH} -I${BATTLEINCL} ${BATTLEPATH}/filters.c

# SODEROBJS -
interfaceB.o: ${TCHINCL}/tchsim.h ${BERGPATH}/application.h ${BATTLEINCL}/globals.h ${BATTLEPATH}/interfaceB.c
cc -c -I${BERGPATH} -I${TCHINCL} -I${BATTLEINCL} -I/usr/fac/hartrum/tchsim/ver2 ${BATTLEPATH}/interfaceB.c

dll.o: ${SODPATH}/dll.c
cc -c -I${SODPATH} ${SODPATH}/dll.c
```

This Makefile generates two executable files called `host` and `battlesim`. 'Host' was designed to be hardware platform-specific, and is always run first; its purpose is to load an application (in this case BATTLESIM) onto each Hypercube node following SPECTRUM after it knows what resources are required. The user specifies the application name, command-line arguments if desired, the number of cube nodes, and the number of LPs in the application. The executable file 'battlesim' is passed as the application to 'host'. Once this information is provided, 'host' loads all of the nodes with its own copy of SPECTRUM and BATTLESIM, in order from LP0 to the number requested, sending

messages to the user's console indicating the progress of loading each LP. Figure 19 depicts the first few lines from a run of the `host` program which uses 4 nodes.

In figure 19, notice that 'host' indicates the current usage of the Hypercube before starting this BATTLESIM scenario; this allows the user to terminate this run if insufficient resources are available. The application name provided to 'host' is **`battlesim`**, the name of the executable file containing the BATTLESIM application. The 'natural' node assignment refers to assigning LP0 to node 0, LP1 to node 1, LP2 to node 2, etc. and is normally used. When loading each node in the scenario, the command-line arguments passed to it are displayed. Thus, this particular invocation properly indicates that multiple scenario files beginning with 'bench13' and the map file 'battlesim13.map' are to be used. Since the '-H' switch is used, all nodes will load the appropriate files and wait for the 'startcube' command to begin exection. This is done to ensure each node's copy of BATTLESIM begins execution at the same time so accurate timing results can be obtained.

```
c386 107:host
Which application do you want to use?:battlesim
Enter the command line arguments for the program (RETURN if none):
>-mbench13 -pbattlesim13
Is assignment of logical processes to nodes to be from a file? (y/n) -> n

The cube is being used as follows:
CUBENAME        USER     SRM           HOST          TYPE            TTYS
iocube          root     cube386       cube386       0
nullwash        pvanhor  cube386       cube386       2m12sxn0        null
How many cube nodes do you want to use? (0 to ABORT):4
How many LP's are in this application?:8
Do you want to use the 'natural' node assignment? (y/n): y
Getting cube of size 4 - stand by.
load -H -p 0 0 battlesim -mbench13 -pbattlesim13
load -H -p 0 1 battlesim -mbench13 -pbattlesim13
load -H -p 0 2 battlesim -mbench13 -pbattlesim13
load -H -p 0 3 battlesim -mbench13 -pbattlesim13
load -H -p 1 0 battlesim -mbench13 -pbattlesim13
load -H -p 1 1 battlesim -mbench13 -pbattlesim13
load -H -p 1 2 battlesim -mbench13 -pbattlesim13
load -H -p 1 3 battlesim -mbench13 -pbattlesim13
startcube
Cube Loaded
Output printing turned off.
Output printing turned off.
         .
         .
         .
```

Figure 19. Example of BATTLESIM execution

E.3 Screen Output.

Once the initial queries from the 'host' program displayed in figure 19 have been answered, the output from BATTLESIM itself is output to the screen. The screen output should appear like the following listing.

```
c386 107:host
Which application do you want to use?:battlesim
Enter the command line arguments for the program (RETURN if none):
>-mbench13 -pbattlesim13
Is assignment of logical processes to nodes to be from a file? (y/n) -> n

The cube is being used as follows:
CUBENAME        USER    SRM           HOST          TYPE            TTYS
iocube          root    cube386       cube386       0
nullwash        pvanhor cube386       cube386       2m12sxn0        null
How many cube nodes do you want to use? (0 to ABORT):4
How many LP's are in this application?:8
Do you want to use the 'natural' node assignment? (y/n): y
Getting cube of size 4 - stand by.
load -H -p 0 0 battlesim -mbench13 -pbattlesim13
load -H -p 0 1 battlesim -mbench13 -pbattlesim13
load -H -p 0 2 battlesim -mbench13 -pbattlesim13
load -H -p 0 3 battlesim -mbench13 -pbattlesim13
load -H -p 1 0 battlesim -mbench13 -pbattlesim13
load -H -p 1 1 battlesim -mbench13 -pbattlesim13
load -H -p 1 2 battlesim -mbench13 -pbattlesim13
load -H -p 1 3 battlesim -mbench13 -pbattlesim13
startcube
Cube Loaded
Output printing turned off.
Output printing turned off.
Output printing turned off.
Output printing turned off.
Output printing turned off.
Output printing turned off.
Output printing turned off.
Output printing turned off.
LP: 1, Input Scenario Read_Time: 0.828000
LP: 2, Input Scenario Read_Time: 0.819000
LP: 7, Input Scenario Read_Time: 0.853000
LP: 4, Input Scenario Read_Time: 0.845000
LP: 6, Input Scenario Read_Time: 0.842000
LP: 5, Input Scenario Read_Time: 0.873000
LP: 3, Input Scenario Read_Time: 0.871000
LP: 0, Input Scenario Read_Time: 2.082000
LAST_TIME message from LP 1 on node 1, pid 0.
LAST_TIME message from LP 2 on node 2, pid 0.
LAST_TIME message from LP 7 on node 3, pid 1.
```

```
LAST_TIME message from LP 4 on node 0, pid 1.
LAST_TIME message from LP 6 on node 2, pid 1.
LAST_TIME message from LP 5 on node 1, pid 1.
LAST_TIME message from LP 3 on node 3, pid 0.
LP 0: Player 1 REACHED_TURNPT at time 0.00 X = 8775.00, Y = 110133.33
LP 0 SENSOR_CHECK: Player 1 being processed.
SENSOR_CHECK: Player 1 has FRONT_END_SENSOR event.
LP 0: Player 1 FRONT_END_SENSOR at time 14.63 X = 23400.00, Y = 110133.33
LP 0 SENSOR_CHECK: Player 1 being processed.
SENSOR_CHECK: Player 1 has CENTER_OF_MASS event.
LP 0: Player 1 CENTER_OF_MASS at time 20.48 X = 29250.00, Y = 110133.33
LP 0 SENSOR_CHECK: Player 1 being processed.
SENSOR_CHECK: Player 1 has BACK_END_SENSOR event.
LP 0: Player 1 BACK_END_SENSOR at time 26.33 X = 35100.00, Y = 110133.33
LP 0 SENSOR_CHECK: Player 1 being processed.
SENSOR_CHECK: Player 1 has FRONT_END_SENSOR event.
LP 0: Player 1 FRONT_END_SENSOR at time 43.88 X = 52650.00, Y = 110133.33
LP 0 SENSOR_CHECK: Player 1 being processed.
SENSOR_CHECK: Player 1 has CENTER_OF_MASS event.
LP 0: Player 1 CENTER_OF_MASS at time 49.73 X = 58500.00, Y = 110133.33
LP 0 SENSOR_CHECK: Player 1 being processed.
SENSOR_CHECK: Player 1 has BACK_END_SENSOR event.
LP 0: Player 1 BACK_END_SENSOR at time 55.58 X = 64350.00, Y = 110133.33
LP 0 SENSOR_CHECK: Player 1 being processed.
SENSOR_CHECK: Player 1 has FRONT_END_SENSOR event.
LP 0: Player 1 FRONT_END_SENSOR at time 73.13 X = 81900.00, Y = 110133.33
LP 0 SENSOR_CHECK: Player 1 being processed.
SENSOR_CHECK: Player 1 has CENTER_OF_MASS event.
LP 0: Player 1 CENTER_OF_MASS at time 78.97 X = 87750.00, Y = 110133.33
LP 0 SENSOR_CHECK: Player 1 being processed.
SENSOR_CHECK: Player 1 has BACK_END_SENSOR event.
LP 0: Player 1 BACK_END_SENSOR at time 84.82 X = 93600.00, Y = 110133.33
LP 0 SENSOR_CHECK: Player 1 being processed.

        .

        .

        .

        .

LP 0: Player 1 FRONT_END_SENSOR at time 764.60 X = 96525.00, Y = 53150.00
LP 0 SENSOR_CHECK: Player 1 being processed.
SENSOR_CHECK: Player 1 has CENTER_OF_MASS event.
LP 0: Player 1 CENTER_OF_MASS at time 770.45 X = 96525.00, Y = 59000.00
LP 0 SENSOR_CHECK: Player 1 being processed.
SENSOR_CHECK: Player 1 has BACK_END_SENSOR event.
LP 0: Player 1 BACK_END_SENSOR at time 776.30 X = 96525.00, Y = 64850.00
LP 0 SENSOR_CHECK: Player 1 being processed.
SENSOR_CHECK: Player 1 has REACHED_TURNPOINT event.
LP 0: Player 1 REACHED_TURNPT at time 813.72 X = 96525.00, Y = 102266.67
LP 0 SENSOR_CHECK: Player 1 being processed.
LAST_TIME message from LP 0 on node 0, pid 0.
```

```
End stats messages:
LP 0 (node 0, pid 0):   0 received,     0 sent.
Max message count set at 10, Max messages removed was 0.
LP 1 (node 1, pid 0):   0 received,     0 sent.
Max message count set at 10, Max messages removed was 0.
LP 2 (node 2, pid 0):   0 received,     0 sent.
Max message count set at 10, Max messages removed was 0.
LP 3 (node 3, pid 0):   0 received,     0 sent.
Max message count set at 10, Max messages removed was 0.
LP 4 (node 0, pid 1):   0 received,     0 sent.
Max message count set at 10, Max messages removed was 0.
LP 5 (node 1, pid 1):   0 received,     0 sent.
Max message count set at 10, Max messages removed was 0.
LP 6 (node 2, pid 1):   0 received,     0 sent.
Max message count set at 10, Max messages removed was 0.
LP 7 (node 3, pid 1):   0 received,     0 sent.
Max message count set at 10, Max messages removed was 0.
LP: 0, stop display executed
HOST: Total CPU time waiting: 0.000000 (msecs)
HOST: Wall clock time loading cube: 19 (secs)
HOST: Wall clock time waiting: 5 (secs)
c386 108:
```

Notice that in the BATTLESIM screen output just listed, the time at which each LP read its scenario is listed, along with the LP identifier. As mentioned earlier, the scenario begins execution after all LPs have read in their own copies of the BATTLESIM application and data files.The LAST_TIME messages after the scenario read times indicate that LPs 1 through 7 have no player events in their respective next event queues, and are done with their portion of the run. This is expected because LPs 1 through 7 have control over no sectors in the battlefield and therefore have no way of gaining responsibility or access to any players. All player events for this run are generated by LP0 only as the set of upcoming messages suggests.

Each time LP0 executes an event, three things occur:the sensor_check function indicates that the next event for a player on LP0 is being determined, it shows what event has been picked for that player, and then it indicates the updated position of the player after the event has been executed (in this case player1). Since Benchmark Scenario 13 uses all three boundary-crossing events, boundary-crossing events *for the same boundary and player* always occur in the order:

1. Front End Sensor
2. Center of Mass
3. Back End Sensor

Looking at the screen output, the user can verify that this is indeed what is happening. You can follow the events, which occur at points identified as small circles, in Figure 18. Player 1, starting in sector 1, travels through every sector in both the positive and negative x/y directions before coming to its final routepoint in sector 4. Once the player has reached its last routepoint, no more player events remain on LP0, so it outputs a LAST_TIME message like the other nodes did. The last step performed by BATTLESIM is to output the ending statistics.

E.4 Log Files.

Log files are identified by their LP identifier concatenated to the word 'log', e.g. the log file belonging to LP0 is log0. The contents of log files are very similar to those output to the user's screen display, but are designed to show more detail than what the user usually wants to have on an interactive basis. The contents of the file log0 are shown next.

```
Initialization.


DET_BOUNDARY_EVENT: Player 1 being processed.

x-axis boundary has FRONT_END_SENSOR crossing
by player 1 at time 14.63


DET_BOUNDARY_EVENT: Player 1 being processed.

x-axis boundary has CENTER_OF_MASS crossing
by player 1 at time 20.48


DET_BOUNDARY_EVENT: Player 1 being processed.

x-axis boundary has BACK_END_SENSOR crossing
by player 1 at time 26.33


DET_BOUNDARY_EVENT: Player 1 being processed.

x-axis boundary has FRONT_END_SENSOR crossing
by player 1 at time 43.88


DET_BOUNDARY_EVENT: Player 1 being processed.

x-axis boundary has CENTER_OF_MASS crossing
by player 1 at time 49.73


DET_BOUNDARY_EVENT: Player 1 being processed.

x-axis boundary has BACK_END_SENSOR crossing
by player 1 at time 55.58


DET_BOUNDARY_EVENT: Player 1 being processed.
```

```
x-axis boundary has FRONT_END_SENSOR crossing
by player 1 at time 73.13


DET_BOUNDARY_EVENT: Player 1 being processed.

x-axis boundary has CENTER_OF_MASS crossing
by player 1 at time 78.97


DET_BOUNDARY_EVENT: Player 1 being processed.

x-axis boundary has BACK_END_SENSOR crossing
by player 1 at time 84.82

          .

          .

          .

          .

y-axis boundary has FRONT_END_SENSOR crossing
by player 1 at time 764.60


DET_BOUNDARY_EVENT: Player 1 being processed.

y-axis boundary has CENTER_OF_MASS crossing
by player 1 at time 770.45


DET_BOUNDARY_EVENT: Player 1 being processed.

y-axis boundary has BACK_END_SENSOR crossing
by player 1 at time 776.30


DET_BOUNDARY_EVENT: Player 1 being processed.

y-axis boundary has FRONT_END_SENSOR crossing
by player 1 at time 823.60


DET_BOUNDARY_EVENT: Player 1 being processed.

Simulation finished.
A total of 60 events were processed.
Max NEQ length was 2.
Final simulation time was 1000.000000.

Initialization time = 3512.000000 msecs.
```

```
Total loop time = 17091.000000 msecs.
Average loop time = 284.850000 msecs.
Termination time = 25.000000 msecs.
Total time = 20628.000000 msecs.

LP 0 wall time taken is 26.843 (secs)
LP 0 messages received 0
LP 0 messages sent 0
```

In log0, only the messages associated with the det_boundary_event routines happened to be activated for this run. The number and kind of messages generated by BATTLESIM for the log files can be tailored to the user's tastes. In this particular case the boundary-crossing messages provide more information than the screen output provided; this extra detail is typical of the messages usually placed in the log file. Lastly, the statistics are shown. The total of 60 events comes from the following breakdown:

- **Two 'End' Events** - These two events are always executed on every node whether any player events were executed or not, and tell both the application and the other nodes that this LP should no longer be considered an active part of the run. However, BATTLESIM recognizes these two events as one single event type known as the END event.
- **Thirteen Reached_Turnpoint Events** - These indicate when player 1 has reached another one of its route points.
- **Fifteen FES events** - Indicate that a player's maximum range front end sensor has crossed another sector boundary.
- **Fifteen COM events** - Indicate that a player's center of mass has crossed another sector boundary.
- **Fifteen BES events** - Indicate that a player's maximum range back end sensor has crossed another sector boundary.

The maximum next event queue length was two, because at any given time in the run it only contained player 1's next event and the LP's END event. The final simulation

time was 1000 milliseconds (msecs) because it is the current default value. The rest of the values vary from run to run, except for message sent and received. Since Benchmark Scenario 13 is trivially parallel, the values for these two fields are zero.

The log files for LP1 through LP7 all contain only the statistics fields shown in log0, since LPs 1-7 did not handle any boundary-crossing events. The contents of one of these log files is now shown.

```
Initialization.


Simulation finished.
A total of 2 events were processed.
Max NEQ length was 1.
Final simulation time was 1002.000000.

Initialization time = 3055.000000 msecs.
Total loop time = 0.000000 msecs.
Average loop time = 0.000000 msecs.
Termination time = 17462.000000 msecs.
Total time = 20517.000000 msecs.

LP 2 wall time taken is 26.356 (secs)
LP 2 messages received 0
LP 2 messages sent 0
```

E.5 Graphics Output File.

The file generated by BATTLESIM for visual display by the DeRouchey graphics driver is called **display.out**. Every time another scenario run is made, BATTLESIM *concatenates, not overlays* the previous contents of **display.out**. Therefore it is very important that the previous graphics output file is deleted before executing scenario run if the user wants to display the scenario. The content of the graphics output file for Benchmark Scenario is now shown.

```
V2.0
terrain.10
32 1 f18
32 2 mig1
32 3 missile
32 4 tank
32 5 truck
30 1 1
50
31 1 0.000000 8775.00 110133.33 500.00 1000.000000 0.000000 0.000000
 90.00000 0.000000 0.000000 0.000000 0.000000 0.000000
31 1 14.625000 23400.00 110133.33 500.00 1000.000000 0.000000 0.000000
 90.000000 0.000000 0.000000 0.000000 0.000000 0.000000
31 1 20.475000 29250.00 110133.33 500.00 1000.000000 0.000000 0.000000
 90.000000 0.000000 0.000000 0.000000 0.000000 0.000000
31 1 26.325000 35100.00 110133.33 500.00 1000.000000 0.000000 0.000000
 90.000000 0.000000 0.000000 0.000000 0.000000 0.000000
31 1 43.875000 52650.00 110133.33 500.00 1000.000000 0.000000 0.000000
 90.000000 0.000000 0.000000 0.000000 0.000000 0.000000
31 1 49.725000 58500.00 110133.33 500.00 1000.000000 0.000000 0.000000
 90.000000 0.000000 0.000000 0.000000 0.000000 0.000000
31 1 55.575000 64350.00 110133.33 500.00 1000.000000 0.000000 0.000000
 90.000000 0.000000 0.000000 0.000000 0.000000 0.000000
31 1 73.125000 81900.00 110133.33 500.00 1000.000000 0.000000 0.000000
 90.000000 0.000000 0.000000 0.000000 0.000000 0.000000
31 1 78.975000 87750.00 110133.33 500.00 1000.000000 0.000000 0.000000
 90.000000 0.000000 0.000000 0.000000 0.000000 0.000000
31 1 84.825000 93600.00 110133.33 500.00 1000.000000 0.000000 0.000000
 90.000000 0.000000 0.000000 0.000000 0.000000 0.000000
31 1 99.450000 108225.00 110133.33 500.00 0.000000 -1000.000000 0.000000
 180.000076 0.000000 0.000000 0.000000 0.000000 0.000000
31 1 144.733330 108225.00 64850.00 500.00 0.000000 -1000.000000 0.000000
 180.000076 0.000000 0.000000 0.000000 0.000000 0.000000
31 1 150.583330 108225.00 59000.00 500.00 0.000000 -1000.000000 0.000000
 180.000076 0.000000 0.000000 0.000000 0.000000 0.000000
31 1 156.433330 108225.00 53150.00 500.00 0.000000 -1000.000000 0.000000
 180.000076 0.000000 0.000000 0.000000 0.000000 0.000000
31 1 201.716660 108225.00 7866.67 500.00 -1000.000000 0.000000 0.000000
```

```
269.999848 0.000000 0.000000 0.000000 0.000000 0.000000
31 1 216.341660 93600.00 7866.67 500.00 -1000.000000 0.000000 0.000000
 269.999848 0.000000 0.000000 0.000000 0.000000 0.000000

                    .

                    .

                    .

                    .

 90.000000 0.000000 0.000000 0.000000 0.000000 0.000000
31 1 716.491660 81900.00 19666.67 500.00 1000.000000 0.000000 0.000000
 90.000000 0.000000 0.000000 0.000000 0.000000 0.000000
31 1 722.341660 87750.00 19666.67 500.00 1000.000000 0.000000 0.000000
 90.000000 0.000000 0.000000 0.000000 0.000000 0.000000
31 1 728.191660 93600.00 19666.67 500.00 1000.000000 0.000000 0.000000
 90.000000 0.000000 0.000000 0.000000 0.000000 0.000000
31 1 731.116660 96525.00 19666.67 500.00 0.000000 1000.000000 0.000000
 359.999924 0.000000 0.000000 0.000000 0.000000 0.000000
31 1 764.599990 96525.00 53150.00 500.00 0.000000 1000.000000 0.000000
 359.999924 0.000000 0.000000 0.000000 0.000000 0.000000
31 1 770.449990 96525.00 59000.00 500.00 0.000000 1000.000000 0.000000
 359.999924 0.000000 0.000000 0.000000 0.000000 0.000000
31 1 776.299990 96525.00 64850.00 500.00 0.000000 1000.000000 0.000000
 359.999924 0.000000 0.000000 0.000000 0.000000 0.000000
31 1 813.716660 96525.00 102266.67 500.00 0.000000 0.000000 0.000000
 359.999924 0.000000 0.000000 0.000000 0.000000 0.000000
86 1000.000000
```

Each of these lines has a special meaning to the DeRouchey graphics display driver. For example, the first line indicates the version of output file, and the second line shows the name of the terrain file for this run. All records after this begin with a record type identifier, with the number and position of fields in the record dependent on the record type. The next five lines identifies the five icons that will be used to represent the five objects in the run: type 1 player are f18's, type 2 players are migl's, and so on. Type 31 records are the most common type of record in the file, because they pass on the updated location of each player to the graphics driver. The first type 31 records contains the state information for player 1 when the run starts; it says:

- Player type is 1 (f18).

- Simulation time is 0.000 msecs.
- X-position on battlefield is 8775 meters.
- Y-position on battlefield is 110133.33 meters.
- Z-position on battlefield is 500 meters.
- X-component of velocity is 1000 meters/second.
- Y-component of velocity is 0 meters/second.
- Z-component of velocity is 0 meters/second.
- Heading is 90 degrees.
- Pitch is 0 degrees.
- Roll is 0 degrees.
- Heading rate of change is 0 degrees/second.
- Pitch rate of change is 0 degrees/second.
- Roll rate of change is 0 degrees/second.

The last record in the file is always a type 86. It tells the DeRouchey graphics driver the time the run ends. For further information on record types, their purpose and the information they expect, the reader should refer to DeRouchey or Rizza's theses (30) (7).

Appendix F. *Detailed Attribute Descriptions*

This appendix describes the updated formats of several data structures in BATTLESIM by explaining the attributes inherent in each one and their purpose. Section F.1 describes the format and attributes in a player, and Section F.2 describes the format and attributes for a scenario input file.

F.1 Player Attribute Description.

The purpose of each of the player attributes is shown in the order they are listed in Figure 8 which depicted the player structure. Note that if an attribute is not actively used in BATTLESIM, then it is marked as such; these attributes remain to support future improvements in simulation fidelity.

- **Sector Identifier** - specifies which sector a player physically resides in.
- **Object Type** - identifies what kind of icon is associated with a player when the scenario's output file is displayed by the graphics display driver.
- **Object Identifier** - unique among "owned" players; however, player-copies of an owned player will share the same object identifier.
- **Object Loyalty** - a number indicating which objects are friends and which are foes. Objects with the same number are friends, and will not attack each other. Objects with different numbers are foes, and may attack one another.
- **Own Player** - indicates whether this player is owned by the sector in which it resides, or if it is just a "player-copy" which is not owned by the sector. ***There can only be one player with a specific player_id which is owned by any sector in the simulation at any time.***
- **Player Size** - attribute indicating the radius of the player. This is used to make collision detection more realistic.
- **Current Time** - current *simulation time* of the player

- **Fuel Status** - value indicates how much fuel the player has left. Not used in the current version.
- **Condition** - value indicating how badly damaged a player is. Not used in the current version, since an object is alway either fully operational or completely destroyed.
- **Vulnerability** - specifies how strong a destructive force is required to destroy the player. Not used in the current version.
- **Location** - three values indicating the position of the player on the battlefield in x,y,and z coordinates.
- **Velocity** - three values indicating the player's x,y, and z velocity vectors.
- **Orientation** - yaw, pitch and roll of player (roll not used in current version).
- **Rotation** - rates of changes about the x, y, and z axis'. These values are not used by the current version.
- **Operator** - two values indicating the experience and threat knowledge of the operator of the player object. Not used in the current version.
- **Performance** - four characteristics of the player not used in the current version. These characteristics include:
    1. max speed
    2. minimum turn radius
    3. average fuel consumption rate
    4. max climb rate
- **Route Data** - a linked list used to hold all the route points of the player in reverse order
- **Sensors** -a linked list used to hold all of the sensors owned by a player. The characteristics of a sensor include:
    1. sensor type (not used by current version)
    2. sensor range

3. sensor resolution (not used by current version)

- **Armaments** - a linked list used to hold all of the armaments owned by a player. The characteristics of an armament include:

  1. type (used by missiles only)
  2. range (not used by current version)
  3. lethality (not used by current version)
  4. accuracy (not used by current version)
  5. speed (not used by current version)
  6. count (not used by current version)

- **Defensive Systems** - a linked list used to hold all of the defensive systems owned by a player. It is not used by the current version. The characteristics of a defensive system include:

  1. type
  2. range
  3. effectiveness

- **Target List** - a linked list used to hold the list of targets for the player. The characteristics of a target list include:

  1. type
  2. location (not used by current version)

- **Player-copies** - a linked list of integers indicating in which sectors *all* the players with the same object_id as this one are located. *This list includes the sector in which this player is located.* This list is required to support the transmission of messages containing player state information updates from the player "owner" to all player "copies" other than itself. Therefore, if this player is not owned by the sector in which it resides, then it will not update the state of players with the same `object_id` in other sectors.

F.2 Scenario Input File Format.

The format for scenario files with the '.in4' extension is shown below. Attributes which existed in the previous version of BATTLESIM and have not changed format are merely listed — additional information on their purpose can be obtained from Soderholm's thesis(33:3-5). New attributes added to support spatial partitioning, as well as old attributes with a new format, are explained in detail.

- **Version Number** - A *character* attribute indicating what version this file is. If the file version does not match that expected by BATTLESIM, then an error message is returned to the user and the run terminates.
- **Terrain Data Filename** - A *character* attribute that indicates the name of the terrain data file to be used for this simulation run. This file contains terrain elevation data which currently is unused by BATTLESIM. However, this was added to support modeling of terrain at a later time.
- **Terrain Min Coordinates (x,y,z)** - Three *double* attributes which provide the minimum x, y, and z axis coordinates of the battlefield. Standardized terrain data files use minimum values of zero.
- **Terrain Max Coordinates (x,y,z)** - Three *double* attributes which provide the maximum x, y, and z axis coordinates of the battlefield. Standardized terrain data files use maximum values of 117,000 for the x-axis and 118,000 for the y-axis. A maximum of 1000 is used for benchmark scenario files in the z-axis.
- **Number of Sectors** - An *integer* value indicating the number of sectors to be used in the scenario. The value must be greater than 0 and less than 65 to be valid.
- **Sector min/max Coordinates** - *Double* values which contain each sector's minimum x/y/z coordinates and maximum x/y/z coordinates, in that order. Each line contains six entries containing the boundary information for a particular sector, with the lines appearing in order from the first to the last sector. Therefore there are exactly as many lines in this section as there are number of sectors.

- **Number of Icon Definitions** - An *integer* value specifying how many icon definitions exist. This value is always five now, since five different types objects can be created in the scenario. This information, while it previously existed, was hardcoded into BATTLESIM.
- **Icon Definitions** - Two attributes, an *integer* and a *character*, which together uniquely describe an icon definition needed to support creation of the display driver datafile using the format previously defined by DeRouchey (7). The five definitions used by current version of BATTLESIM include:
  - Type 1 - f18
  - Type 2 - mig1
  - Type 3 - missile
  - Type 4 - tank
  - Type 5 - truck

  This information, while it previously existed, was hardcoded into BATTLESIM.
- **Player type thru max climb** - Nineteen attributes, all residing on the same line in the scenario file, which provide information about a particular player in the scenario. The attributes, in order, are:
  1. Object Type
  2. Object Identifier
  3. Object Loyalty
  4. Current Time
  5. Fuel Status
  6. Condition
  7. Vulnerability
  8. Velocity (x-component)
  9. Velocity (y-component)

10. Velocity (z-component)
11. Orientation (yaw rate)
12. Orientation (pitch rate)
13. Orientation (roll rate)
14. Operator (experience)
15. Operator (threat knowledge)
16. Performance Characteristics (minimum turn radius)
17. Performance Characteristics (max speed)
18. Performance Characteristics (average fuel consumption rate)
19. Performance Characteristics (max climb rate)

- **Number of Route Points** - An *integer* value specifying how many route points there are in the player's **route_data** linked list.
- **Route Points** - Three *double* values indicating the x,y, and z coordinate of one of the player's route points. The points are listed in the order the player goes to each of them. Each line in the file contains exactly one route point, so there are as many lines as there are route points.
- **Number of Sensors** - An *integer* value specifying how many sensors there are in the player's **sensors** linked list.
- **Sensors** - Three *integer* attributes indicating the type, range, and resolution of one of the player's sensors, respectively. Each line in the scenario file contains one sensor, so there are as many lines a^ there are sensors.
- **Number of Armaments** - An *integer* value specifying how many armaments there are in the player's **armaments** linked list.
- **Armaments** - Six *integer* attributes indicating the type, range, yield, accuracy, speed, and count of one of the player's armaments, in that order. Each line in the scenario file contains one aramament, so there are as many lines as there are armaments.

- **Number of Targets** - An *integer* value specifying how many targets (by type or location) are in the player's **`targets`** linked list.
- **Targets** - One *integer* indicating the type, and three *double* values indicating the x, y, and z coordinate of one of the player's targets, in that order. Each line in the scenario file contains one target, so there are as many lines as there are targets.
- **Number of Defensive Systems** - An *integer* value specifying how many defensive systems are in the player's **`defensive_systems`** linked list.
- **Defensive Systems** - Three *integer* values indicating the type, range, and effectiveness of one of the player's defensive systems, respectively. Each line in the scenario file contains one defensive system, so there are as many lines as there are defensive systems.

Since every player in a scenario must have at least one route point it is starting at as well as trying to reach, no player will have an empty route list at initialization. However, any of the other four player linked lists may be empty. In that case, the linked list in question would have no lines in the scenario file to describe it other than the number attribute. Notice that no information on each player's player-copies is provided in the scenario file — BATTLESIM realizes at initialization time that each player has exactly one player description in the scenario files, and will create and update the linked list automatically.

Appendix G. *Examples of BATTLESIM Events*

This appendix contains examples of center of mass (COM) and back end sensor (BES) events. Section G.1 contains the examples for the COM events, and Section G.2 contains BES examples. Refer to Chapter 5 for complete examples of BATTLESIM front end sensor (FES) events.

G.1 Center of Mass Events.

Figure 20 depicts a valid COM event for sector three. Player one is moving through sector three towards sector four with a positive x velocity component. Player one's center of mass, located at $x_{present}$, has not yet reached the next sector boundary to be crossed at x_{future}.

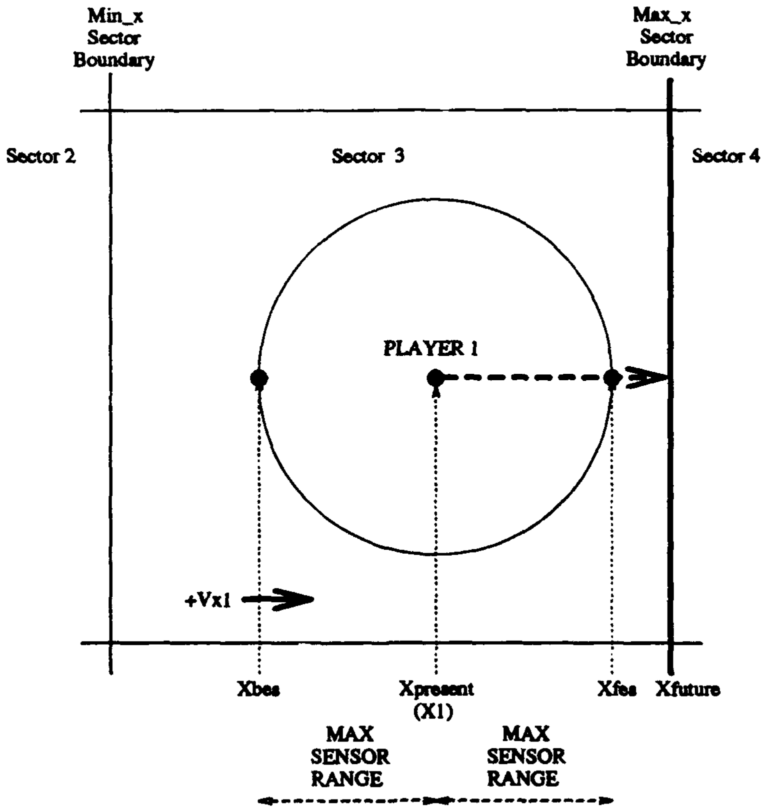

Figure 20. Valid Center Of Mass Sensor Event (Positive x-velocity)

The location of these two entities in sector three is:

$x_{present} = x_1$

x_{future} = sector's maximum x coordinate = max_x

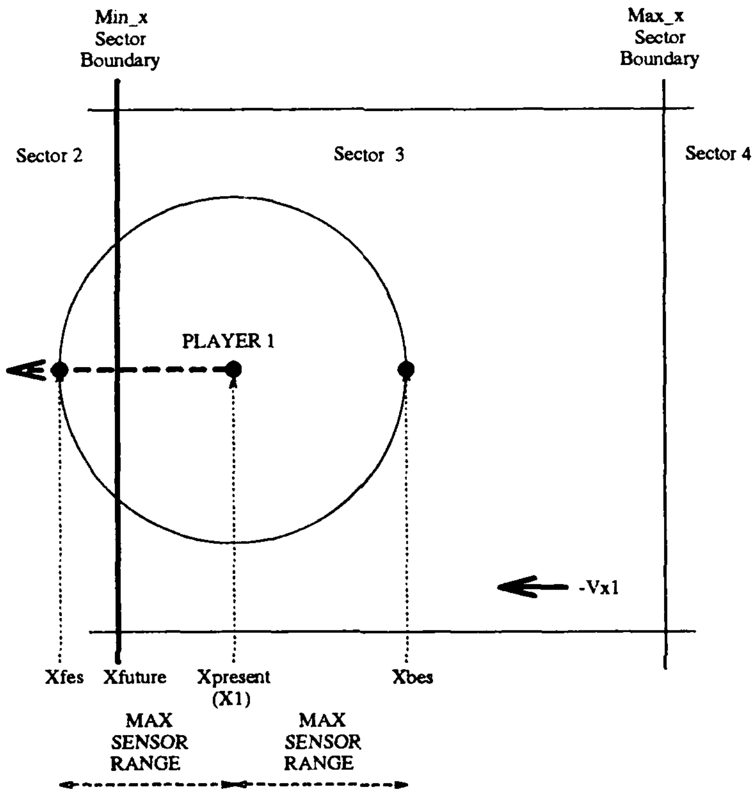

Figure 21. Valid Center Of Mass Sensor Event (Negative x-velocity)

Figure 21 depicts the other possible condition for a valid COM event. Player one is traveling from sector three towards sector two with a negative x velocity. Player one's center of mass has not yet reached the next sector boundary it is traveling towards. The values of $x_{present}$ and x_{future} are:

$x_{present} = x_1$

x_{future} = sector's minimum x coordinate = min_x

The reason these two COM events are valid is not difficult to understand when the conditions for a valid COM event are examined more closely. The player has not yet reached the boundary of the sector in which it is located. If the player with the positive x velocity is already in the next sector, then its present location $x_{present}$ has a larger x-

value than the sector boundary value x_{future}, which results in a negative time value being returned when Equation 5, described in Chapter 5, is applied. Similarly, if the player with a negative x velocity is already in the next sector, then its present location has a smaller x-value than x_{future}, which results in a negative value of time for the COM event. Since $x_{present}$ is greater than x_{future} with a negative x velocity, figure 21 depicts a valid COM event.

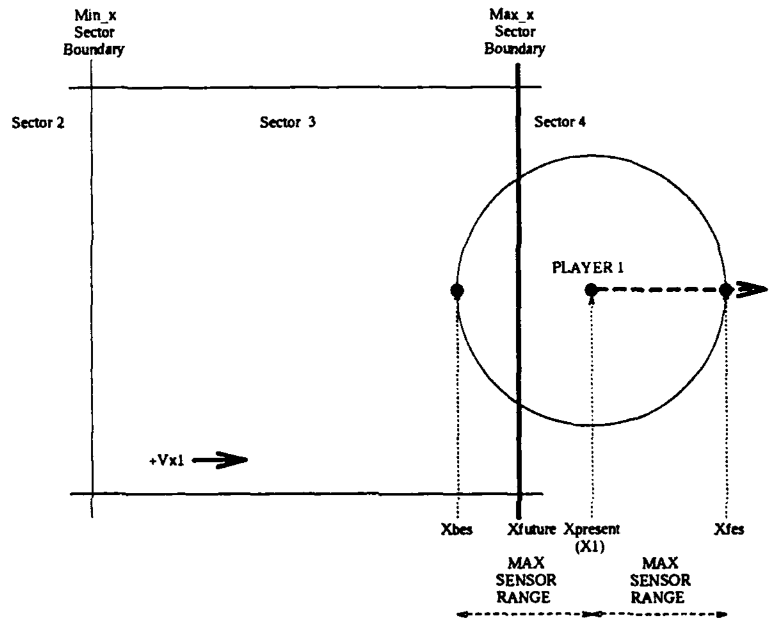

Figure 22. Invalid Center Of Mass Sensor Event (Positive x-velocity)

Figures 22 and 23 show invalid COM events for sector three. Figure 22 is invalid because the value $x_{present}$ is not less than x_{future} with a positive x velocity component, and figure 23 is invalid because the value $x_{present}$ is not greater than the value of x_{future} with a negative x velocity component.

G.2 Back End Sensor Events.

Figure 24 shows a valid BES event for sector three. Player one is traveling out of sector three towards sector four, and player one's back end sensor, located at $x_{present}$, has not reached its next sector boundary at x_{future}. It is very important to understand that *this diagram is determining whether a valid BES event exists for sector three even though*

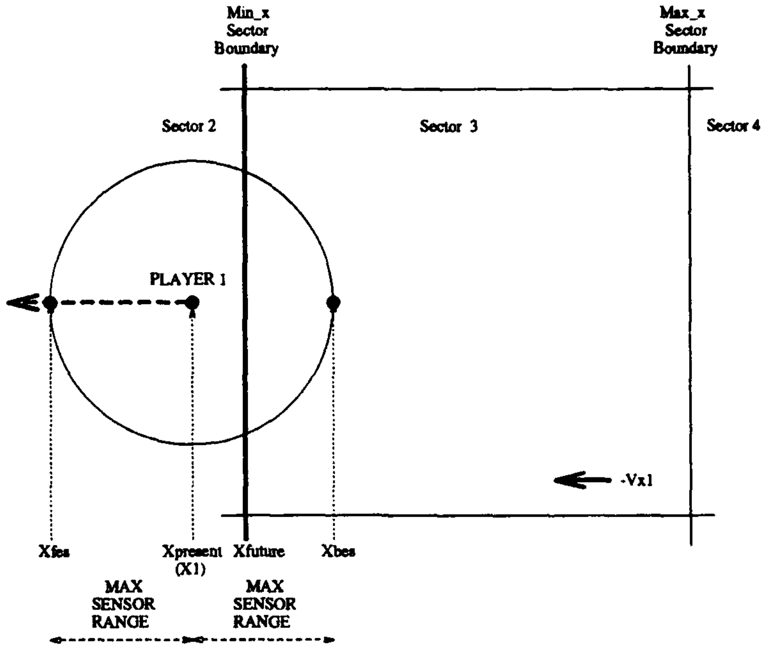

Figure 23. Invalid Center Of Mass Sensor Event (Negative x-velocity)

the player is already in sector four, because a player's center of mass is transferred from one sector upon completion of the COM event — a stated prerequisite event for the BES event. Since diagram 24 depicts a BES event with a positive x velocity component, the values for $x_{present}$ and x_{future} are:

$x_{present} = x_1$ - max_sensor_range

x_{future} = last sector's max_x = min_x of sector now containing player

The use of these two values in Equation 5, described in Chapter 5, results in the expression:

$$t = \frac{x_{future} - x_{present}}{v_{x1}} = \frac{\text{min\_x} - (x_1 - \text{ max\_sensor\_range})}{v_{x1}}$$

Figure 25 illustrates the second way of generating a valid BES event, in this case with a negative x component velocity. Player one is traveling through sector three away from sector four. However, the back end sensor of player one has not yet encountered the sector boundary. The values for $x_{present}$ and x_{future} in this diagram are:

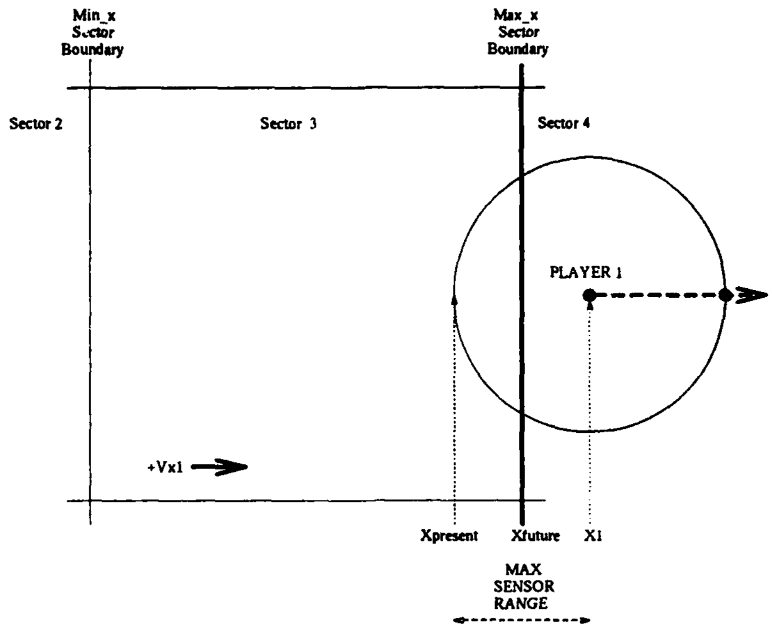

Figure 24. Valid Back End Sensor Event (Positive x-velocity)

$x_{present} = x_1 + \text{max\_sensor\_range}$

$x_{future} = \text{last sector's min\_x} = \text{max\_x of sector now containing player}$

The use of these two values in Equation 5 results in the expression:

$$t = \frac{x_{future} - x_{present}}{v_{x1}} = \frac{max\_x - (x_1 + \text{ max\_sensor\_range})}{v_{x1}}$$

Since $x_{present}$ is greater than x_{future}, this is also a valid BES sensor event.

Figures 26 and 27 illustrate invalid BES events for player one in sector three. These events are invalid because in both diagrams player one's back end sensor, located at $x_{present}$, has already traveled past sector three's boundary at Therefore, while the values for $x_{present}$ and x_{future} have changed, the same relationships stated between them in the other algorithms is still maintained.

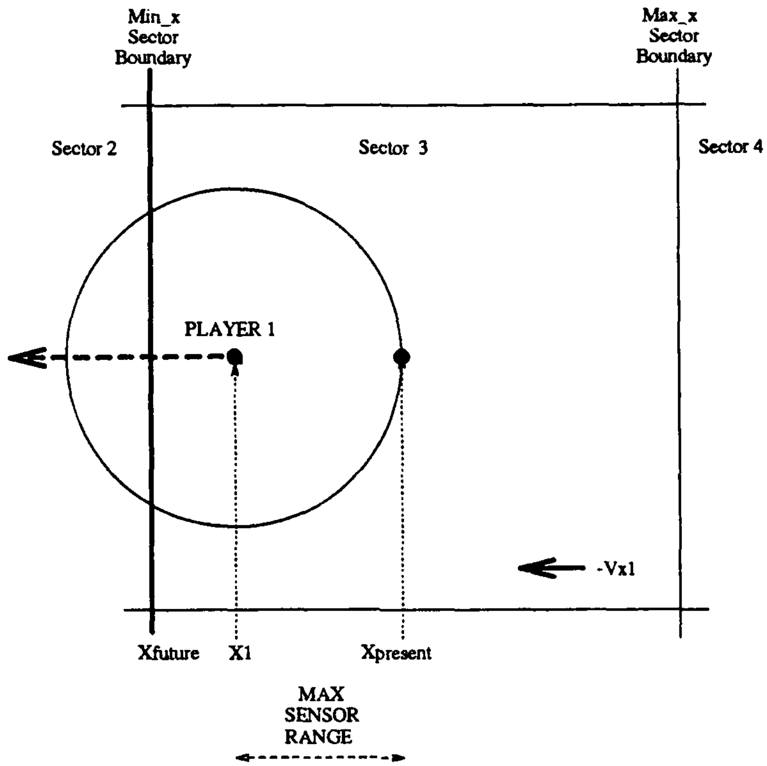

Figure 25. Valid Back End Sensor Event (Negative x-velocity)

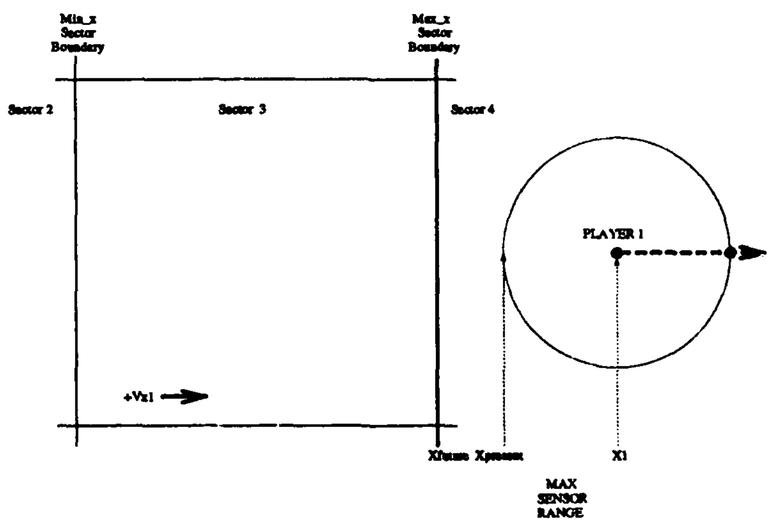

Figure 26. Invalid Back End Sensor Event (Positive x-velocity)

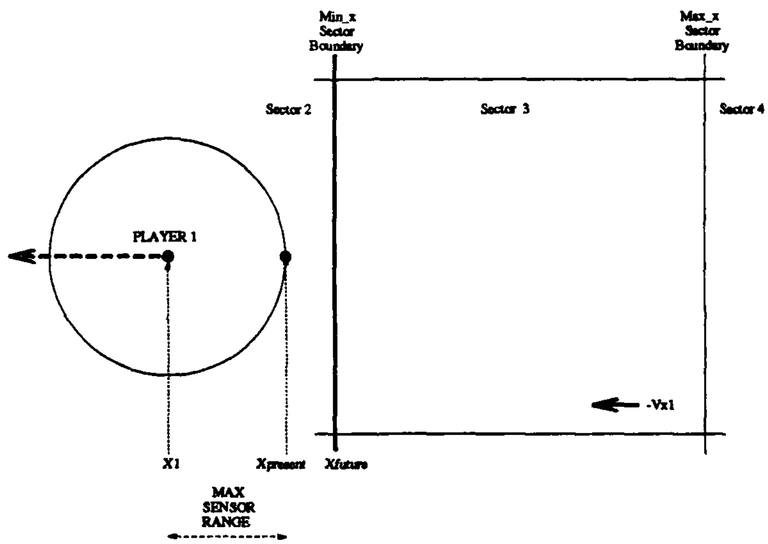

Figure 27. Invalid Back End Sensor Event (Negative x-velocity)

Bibliography

1. Booch, Grady. *Software Engineering with Ada* (Second edition Edition). The Benjamin/Cummings Publishing Company, Inc., 1986.

2. Booch, Grady. *Software Components with Ada: Structures, Tools, and Subsystems*. The Benjamin/Cummings Publishing Company, Inc., 1987.

3. Breeden, Thomas A. *Parallel Implementation of VHDL Simulations on the Intel iPSC/2 Hypercube*. MS thesis, AFIT/GCE/ENG/92D-01, School of Engineering, Air Force Institute of Technology (AU), Wright-Patterson AFB OH, December 1992.

4. Bryant, R.E. *Simulation of Packet Communication Architecture Computer Systems*. Technical Report, Cambridge MA: Massachussetts Institute of Technology, 1977. Technical Report MIT, LCS, TR-188.

5. Chandy, K.M. and J. Misra. "Asynchronous Distributed Simulation via a Sequence of Parallel Computations," *The Association for Computing Machinery*, 198–206 (1981).

6. Daniels, David W. *Development of a Hardware Acceleration Engine*. MS thesis, AFIT/GCE/ENG/93M-01, School of Engineering, Air Force Institute of Technology (AU), Wright-Patterson AFB OH, December 1992.

7. DeRouchey, William. *A Remote Visual Interface Tool for Simulation Control and Display*. MS thesis, AFIT/GCS/ENG/90D-03, School of Engineering, Air Force Institute of Technology (AU), Wright-Patterson AFB OH, December 1990.

8. Fujimoto, Richard M. "Peformance Measurements of Distributed Simulation Strategies." *Proceedings of the SCS Multiconference on Distributed Simulation*. 14–20. San Diego: The Society for Computer Simulation International, 1988.

9. Fujimoto, Richard M. "Parallel Discrete Event Simulation." Earlier version approved in the ACM Winter Simulation Conference, 1989.

10. Fujimoto, Richard M. "Performance of Time Warp Under Synthetic Workloads." *Proceedings of the SCS Multiconference on Distributed Simulation*. 23–28. San Diego: The Society for Computer Simulation International, 1990.

11. Gafni, Anat. "Rollback Mechanisms for Optimistic Distributed Simulation Systems," *Distributed Simulation*, 61–67 (1988).

12. Hartrum, Thomas C. and Brian J. Donlan. "HYPERSIM: Distributed Discrete-Event Simulation on an iPSC," *The Association for Computing Machinery*, 745–747 (1988).

13. Hatrum, Thomas C. "AFIT Guide to SPECTRUM." February 1992.

14. Hatrum, Thomas C. "Project Q: A Case Study in Parallel Discrete Event Simulation." May 1992.

15. Hatrum, Thomas C. "TCHSIM: A Simulation Environment for Parallel Discrete Event Simulation." April 1992.

16. Jefferson, David. "Virtual Time," *The Assocation for Computing Machinery*, 404–424 (1985).

17. Jefferson, David. "Distributed Simulation and the Time Warp Operating System," *The Assocation for Computing Machinery*, 77–93 (1987).

18. Jefferson, David. "Virtual Time II: Storage Management in Distributed Simulation." December 1989.

19. Lin, Yi-Bing and Edward D. Lazowska. "Optimality Consideratoins of Time Warp Parallel Simulation." *Proceedings of the SCS Multiconference on Distributed Simulation*. 29–34. San Diego: The Society for Computer Simulation, 1990.

20. McDonald, Bruce, "Distributed Interactive Simulation: Operational Concept(DRAFT)," February 1992. Prepared for PM Trade by UCF Institute for Simulation & Training.

21. McDonald, Bruce, "Distributed Interactive Simulation: Standards Development Guidance Document(DRAFT)," February 1992. Prepared for PM Trade by UCF Institute for Simulation & Training.

22. Moser, Robert S. *A Spatially Partitioned Parallel Simulation of Colliding Objects*. MS thesis, AFIT/GCS/ENG/91D-15, School of Engineering, Air Force Institute of Technology (AU), Wright-Patterson AFB OH, December 1991.

23. Nandy, Biswajit and Wayne M. Loucks. "An Algorithm for Partitioning and Mapping Conservative Parallel Simulation onto Multicomputers." *Proceeding of the 1992 SCS Western Simulation MultiConference on Parallel and Distributed Simulation*. 139–146. San Diego: The Society for Computer Simulation, 1992.

24. Nicol, David M. "Mapping a Battlefield Simulation onto Message-passing Parallel Architectures." *Proceedings of the SCS Multiconference on Distributed Simulation*. 141–146. San Diego: The Society for Computer Simulation International, 1988.

25. Preiss, Bruno R. et al. "On the Trade-Off between Time and Space in Optimistic Parallel Discrete-Event Simulation." *Proceeding of the SCS Western Simulation MultiConference on Parallel and Distributed Simulation*. 33–42. San Diego: The Society for Computer Simulation, 1992.

26. Reiher, Peter and others. "Cancellation Strategies in Optimistic Execution Systems." *Proceedings of the SCS Multiconference on Distributed Simulation*. 112–121. San Diego: The Society for Computer Simulation, 1990.

27. Reiher, Peter L. and David Jefferson. "Virtual Time Based Dynamic Load Management in the Time Warp Operating System." *Proceedings of the SCS Multiconference on Distributed Simulation*. 103–111. San Diego: The Society for Computer Simulation, 1990.

28. Reynolds, Jr., Paul F. "A Spectrum of Options for Parallel Simulation." *Proceedings of the ACM Winter Simulation Conference*. 1988.

29. Reynolds, Jr., Paul F. et al. "Comparative Analyses of Parallel Simulation Protocols." *Proceedings of the ACM Winter Simulation Conference*. 671–679. 1989.

30. Rizza, Robert J. *An Object-Oriented Military Simulation Baseline for Parallel Simulation Research*. MS thesis, AFIT/GCS/ENG/90D-12, School of Engineering, Air Force Institute of Technology (AU), Wright-Patterson AFB OH, December 1990.

31. Ross, D.T. et al. "Software Engineering: Process, Principles, and Goals," *Computer*, 65–67 (1975).

32. Schuppe, Thomas F. "Modeling and Simulation: A Department of Defense Critical Technology." *Proceedings of the 1991 Winter Simulation Conference*. 519–525. San Diego: The Society for Computer Simulation International, 1991.

33. Soderholm, Steven R. *A Hybrid Approach to Battlefield Parallel Discrete Event Simulation*. MS thesis, AFIT/GCS/ENG/91D-23, School of Engineering, Air Force Institute of Technology (AU), Wright-Patterson AFB OH, December 1991.

34. Sokol, Lisa M. et al. "MTW: A Strategy for Scheduling Discrete Simulation Events for Concurrent Execution." *Proceedings of the SCS Multiconference on Distributed Simulation*. 34–42. San Diego: The Society for Computer Simulation International, 1988.

35. Su, Wen-King and Charles L. Seitz. "Variants of the Chandy-Misra-Bryant Distributed Discrete-event Simulation Algorithm." *Proceedings of the SCS Multiconference on Distributed Simulation*. 38–43. San Diego: The Society for Computer Simulation International, 1989.

36. Thesen, Arne and Laurel E. Travis. "Introduction to Simulation." *Proceedings of the 1991 Winter Simulation Conference*. 5–14. San Diego: The Society for Computer Simulation International, 1991.

37. VanHorn, Prescott J. *Development of a Protocol User's Guideline for Conservative Parallel Simulations*. MS thesis, AFIT/GCS/ENG/92D-19, School of Engineering, Air Force Institute of Technology (AU), Wright-Patterson AFB OH, December 1992.

38. Wagner, David B. et al. "Techniques for Efficient Shared-memory Parallel Simulation." *Proceedings of the SCS Multiconference on Distributed Simulation*. 29–37. San Diego: The Society for Computer Simulation International, 1989.

39. Wieland, Frederick and David Jefferson. "Speedup Bias." DRAFT, 1989.

40. Wieland, Frederick et al. "Distributed Combat Simulation and Time Warp: The Model and its Performance." *Proceedings of the SCS Multiconference on Distributed Simulation*. 14–20. San Diego: The Society for Computer Simulation International, 1989.

41. Wieland, Frederick et al. "An Empirical Study of Data Partitioning and Replication in Parallel Simulation." *Proceeding of The Fifth Distributed Memory Computing Conference*. 915–921. Los Alamitos: IEEE Computer Society Press, 1990.

42. Zeigler, Bernard P. "Object-Oriented Modelling and Discrete Event Simulation." Draft copy of paper in Vol 33 Advances in Computers, edited by Marshall Yovits, August 1991.

Vita

Captain Kenneth C. Bergman was born August 10, 1964, in Washington D.C. After graduating from Wabash High School in 1982, he enrolled in Rose-Hulman Institute of Technology. In May 1986, he graduated with a Bachelor of Science in Computer Science degree and an Air Force ROTC commission as a second lieutenant. Entering active duty in January 1987, Capt Bergman worked as a software specialist for operating location AB (OL-AB) based at Peterson Air Force Base, Colorado; the unit was responsible for the formation of computer modules at Falcon Air Force Base, Colorado to control Department of Defense satellites. Capt Bergman subsequently worked as the manager of a satellite operations training module before his entry into AFIT in June 1991.

Permanent address: 511 Washington St.
Wabash IN 46992

REPORT DOCUMENTATION PAGE

December 1992 Master's Thesis

SPATIAL PARTITIONING OF A BATTLEFIELD
PARALLEL DISCRETE-EVENT SIMULATION

Kenneth C. Bergman, Captain, USAF

Air Force Institute of Technology, WPAFB OH 45433-6583

AFIT/GCS/ENG/92D-03

Major Dave Neyland
DARPA (Advanced Technology Simulation Office)
3701 North Fairfax Drive
Arlington, VA 22203
(703) 696-2298

Approved for public release; distribution unlimited

This thesis describes a method for spatially partitioning a battlefield into units known as sectors to achieve speedup two ways: through the reduction of each battlefield object's next event search space, and lowering the amount of message-passing required. Each sector is responsible for tracking and controlling access to all objects within its boundaries. A distributed proximity detection algorithm employing boundary-crossing events is used to control player movement between sectors. Each object's state information is replicated in all sectors it has sensor capability for the minimum time required; this ensures that each object's next event is properly determined based upon interactions with objects in other sectors as well as its own.

Each scenario is initialized using three sources of information: a set of scenario input files, a mapping file, and command-line arguments. Scenarios generate output in the form of screen messages, log files, and graphics display files.

The issues involved in determining when and how to dynamically change the boundaries are discussed. A heuristic for changing sector boundaries based upon the number of players in each sector, as well as player attributes, is proposed.

Battlefield, Parallel, Discrete-Event, Simulation, Spatial partitioning, Object-based

188

UNCLASSIFIED UNCLASSIFIED UNCLASSIFIED UL